NEUROCUBE: ENERGY-EFFICIENT PROGRAMMABLE DIGITAL DEEP LEARNING ACCELERATOR BASED ON PROCESSOR IN MEMORY PLATFORM

A Dissertation
Presented to
The Academic Faculty

By

Duckhwan Kim

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

December 2017

NEUROCUBE: ENERGY-EFFICIENT PROGRAMMABLE DIGITAL DEEP LEARNING ACCELERATOR BASED ON PROCESSOR IN MEMORY PLATFORM

Approved by:

Dr. Saibal Mukhopadhyay, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Asif Islam Khan
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sek Chai
Center for Vision Technologies
SRI International

Date Approved: August 3, 2017

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NeuroCube: Energy-Efficient Programmable Digital Deep Learning Accelerator Based on Processor In Memory Platform

Duckhwan Kim

122 Pages

Directed by Dr. Saibal Mukhopadhyay

Deep learning, machine learning algorithm based on artificial neural network, shows great success in numerous pattern recognition problems, such as image recognition or speech recognition. Most of deep learning developments are based on the software platform with general purpose graphic processor units (GPU). In terms of efficiency, however, operating deep learning with GPU is limited by power/thermal budget to be operated in mobile device or high performance computing cluster. In this thesis, I present a programmable and scalable deep learning accelerator based on 3D high-density memory integrated with logic tier. The proposed architecture consists of clusters of processing engines (PEs) and the PE clusters access multiple memory channels (vaults) in parallel. The operating principle, referred to as the memory centric computing, embeds specialized state-machines within the vault controllers of HMC to drive data into the PE clusters. Next version of NeuroCube is designed to improve throughput of global connections (fully connections) in the deep neural network, which is critical in recurrent neural network (RNN). NeuroCube is changed to accelerate deep learning training, which requires additional optimized data flow to improve throughput for both inference and training. For computing gradient, it also supports 32bit fixed point with stochastic rounding to prevent gradient vanishing issue. A programming model and supporting architecture utilizes the flexible data flow to efficiently accelerate training of various types of DNNs. The cycle level simulation and synthesized design in 15nm FinFET shows power efficiency of 500 GFLOPS/W, and almost similar throughput for a wide range of DNNs including convolutional, recurrent, multi-layer-perceptron, and mixed (CNN+RNN) networks.

CHAPTER 1
INTRODUCTION

Deep learning, application to learn given functions (benchmark) based on neuro-inspired network (deep neural network: DNN) with massive training data, shows a great success in classical image recognition [1, 2, 3], robotics path planning [4], natural language processing [5, 6, 7], and even politic speech generation [8] recently.

Deep learning is composed of two phases: training and inference. Training is searching a set of parameters of the deep neural network with massive training data composed of inputs (ex: image) and its desired outputs (ex: label of the object in image classification). Using discrete optimization algorithm such as gradient descent [9, 10, 11], parameters are updated during the iteration in training to minimize the cost function of the deep neural network (error at the final output). Inference is estimating output for the new input, which is not trained before, based on the trained network.

In terms of deep learning accelerator design, computational efficiency is a primary concern for both inference and training. Recently, there are demands for inference in real time applications by IoT devices with limited power budget [12, 13]. In training, computational efficiency is more critical. Although it is performed in high performance computing powered by general purpose graphic processor unit (GPGPU) with many parallel cores, with stable power issue, thermal issue becomes critical in deep leraning training [14].

Over the last decade, the number of the parameters in neural network have continued to grow dramatically, e.g. from $\sim$60K parameters in 1998 for LeNet5 [15], a Convolutional Neural Network (CNN), to over 120M parameters in 2014 for DeepFace, a human face identification [16]. Further, proliferation of autonomous systems increases the demanding of in-field learning requiring hardware platforms capable of supporting on-chip training [17, 18].

This trend inevitably makes training or inference both computationally challenging and memory intensive. Compared to conventional algorithm, DNNs place significant demands on the memory bandwidth due to the low ops/byte ratio in the evaluation of these networks. Although GPU have driven much of the recent success in Deep Learning, its system performance (throughput) is less than 20% of peak throughput for the deep learning training [19].

Researchers are also exploring other computing fabrics such as FPGAs [20, 21, 22], and more recently, ASIC accelerators [20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33] to improve the computational performance and power efficiency of deep learning.

Since deep learning has very low operation density (number of operations per data movement), data movement between memory and processing elements becomes critical, called memory-wall issue. To solve memory-wall issue, recent studies have placed DNN accelerators in the memory and exploited the data flow behavior of DNNs to improve energy efficiency or optimize memory system performance via data reuse in on-chip memory structures. Controlling data flow allows system to be programmable to cover different network connections (fully connected, spatial locally connected, etc.) [27, 28].

In this thesis, the primary objective is to present the energy-efficient programmable digital deep learning accelerator for both inference and training. Important features should be addressed in accelerator design are summarized as below:

Universal DL accelerator

Deep learning is composed of two phases: inference and training. Although training is required only once for given task (ex: image recognition using IoT device), re-training is required to improve better accuracy during the inference (ex: adapting environment where IoT device is placed). Moreover, training with high performance computing unit becomes challenge as well due to thermal and power issue. In this thesis, Neurocube is designed to accelerate both inference and training.

Programmability

Since deep neural network structure can vary based on the input size, application type,

complexity of the application, fixed hardware for specific deep learning is impractical. Moreover, new type of layers in terms of connection are still studied; thus programmability is important feature to accelerate newly developed deep neural network. Similar to GPU/CPU, accelerator should be programmable for each layer or each operation.

Memory optimization

Due to low operation density (Ops/byte), overhead of data movement between computing units and memory should be minimized. To take advantage of many core systems to process DL in parallel, not only memory bandwidth but also memory concurrency are important. To solve memory wall issue, (1) accelerator design with high density on-chip memory such as eDRAM [24], (2) compress or prune the *near zero* parameters to be stored in the on-chip memory[26], or (3) process near memory or process in memory platform are introduced [28, 29, 30].

Energy efficiency

To operate deep learning application in mobile device in real time (ex: face recognition [16]), both low power consumption and low latency for real time application should be achieved. In training, state-of-art GPU consuming more than 3KW can train less than 1,500 images of ImageNet [34] benchmark in a second. Considering total number of images in training set and number of iterations for training, training still needs couple of days for a simple network. To improve energy efficiency, allowing small error during the arithmetic operation has been studied. Thanks to error resiliency of the deep neural network, approximate computing can save dynamic power and area of hardware while maintaining the accuracy.

Fig 1.1 shows basic architecture of NeuroCube in 3D stacked DRAM and its processing efficiency compared to previous work. In both inference and training, NeuroCube shows higher efficiency than previous ASIC architecture and GPU or server level CPU.

Before deep learning architecture design, the impact of quantization error in fixed point on both inference and training will be studied (Chapter 3). Based on the study, multipli-

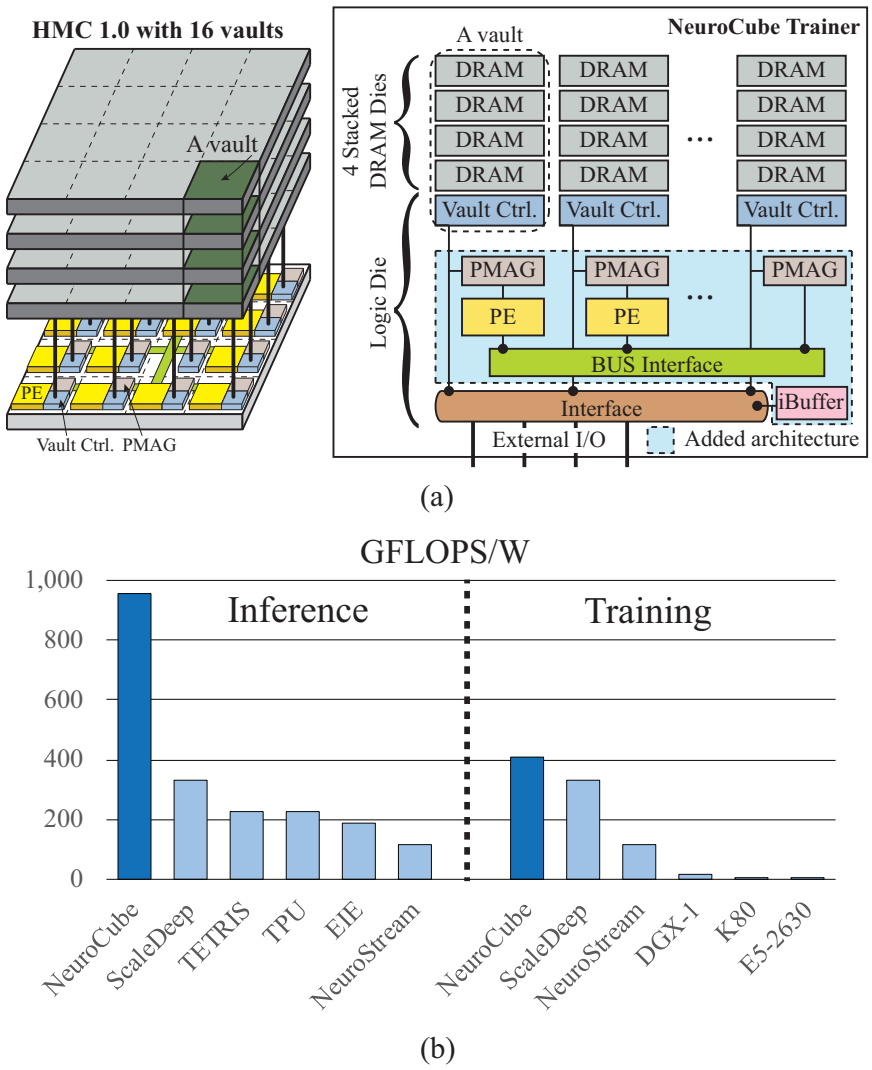


Figure 1.1: NeuroCube overview. (a) NeuroCube architecture as processor in memory and (b) its processing efficiency (GOPS/W).

cation and addition (MAC) unit is designed and placed in NeuroCube, the deep learning accelerator as processor in memory platform within hybrid memory cube (HMC). It is desiged by placing 2D array of processing elements below DRAM dies (Chapter 4). It is improved to accelerate global connection to accelerate fully connected layer, recurrent layer, and sparse global connection (some of synaptic weights are zero) (Chapter 5). At last, NeuroCube, which allows on-chip training and shows high throughput performance for both inference and training, is designed (Chapter 6).

CHAPTER 2

BACKGROUND ON DEEP LEARNING ACCELERATOR

2.1 Deep Learning

Deep learning is the machin learning based on neuro-inspired network (artificial neural network). For given non-linear functions and its input-output pairs, neural network is trained to *approximate* the target function. Recently, it shows great success to represent wide range of applications [1, 2, 3, 4, 5, 6, 7, 8, 35].

In this thesis, I will use the term *neural network* (NN) to represent an artificial neural network [35]. In general, NN is composed of multiple layers of neurons where each layer is composed of multiple neurons. The first layer, which receives the raw input (e.g., image), is called the input layer. The last layer, which generates the output of the NNs, is called the output layer. The multiple layers between the input and the output layers are referred to as the hidden layers.

Fig. 2.1 (a) illustrates the neuron, which state (y_i) is defined as summation of weighted (w_{ik}) inputs (x_k) and the threshold is represented as an activate function ($N.L(y)$). The state (y_i) and output (x_i) of neuron i is computed as

$$y_i = \sum_k w_{ik} \cdot x_k \tag{2.1}$$

$$x_i = N.L(y_i). \tag{2.2}$$

where, $N.L()$ is a non-linear activate function. The basic operation of a single neuron is common to all NNs (Eq. 2.1) while the activate function may differ. Different NNs can be defined by the connections of the neurons in the network. For example, a multi-

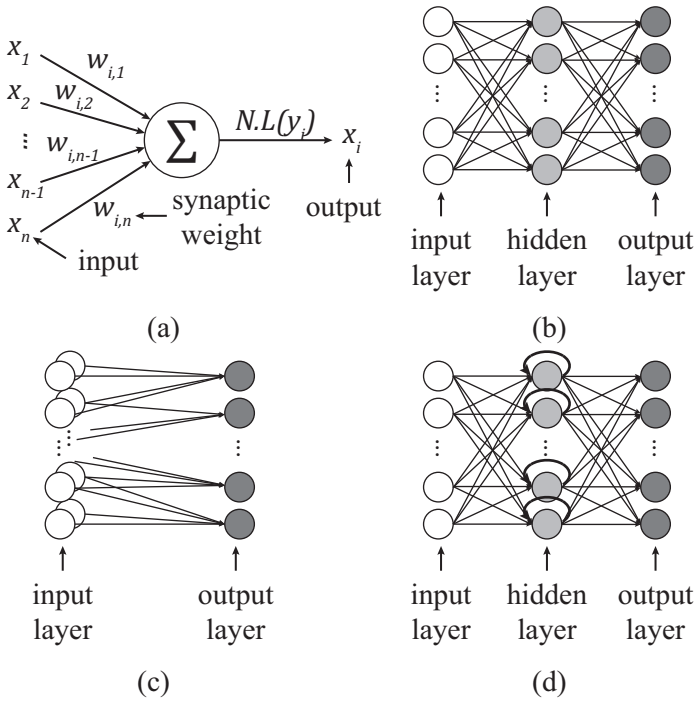


Figure 2.1: (a) Simple neuron diagram, (b) Fully-connected feedforward composed of input layer, one hidden layer, and output layer, (c) Convolutional neural network (feedforward, sparse connection), and (d) Recurrent neural network (fully-connected feedback).

layer perceptron (MLP [36]) is a feedforward network in which each neuron in one layer is connected to all neurons in the next layer (Fig. 2.1 (b)). In Eq. 2.1, k refers to all connected neurons in the preceding layer. For a convolutional neural network (ConvNN [9]), k is the 2D-neighborhood of i; therefore only a few neurons placed locally together are connected to a neuron in next layer (Fig. 2.1 (c)). For a recurrent neural network (RNN [37]), k refers to all neurons in the preceding layer and includes itself (recurrent); the current output is the input to compute the state at the next time step (Fig. 2.1 (d)).

I should note that the main difference between different neural networks is the *set k* - the set of neurons connected to a single neuron in the next layer. In other words, each class of neural networks can be defined by the connectivity between neurons, while the basic operation of a neuron remains the same - (*multiplication and accumulation*). Therefore, I observe that a group of multiply-accumulate (MAC) units can be used to emulate a range of neural network sizes and types by re-programming the connectivity between the units.

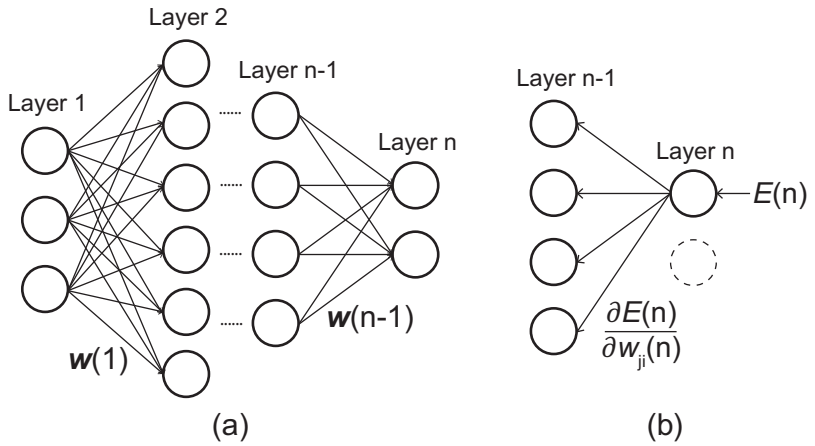


Figure 2.2: (a) The operation of feedforward neural network. (b) The back-propagation training algorithm computing the error sensitivity (∂E/∂w_{ji})

I further observe that programmable connectivity simply means orchestrating the required data flows between memory and the MAC units.

2.1.1 Multiple Layer Perceptron (MLP)

Before moving to state-of-art deep nerual network, I will breifly overview the basic neural network, multiple layer perceptron, and its training using backpropagation.

Multilayer perceptron (MLP) is the simple feedforward network (no recurrent connection) with multiple layers [35]. All equations and notations for this chapter follow [35]. Fig. 2.2 (a) illustrates MLP with multiple layers. The state of each neuron is computed as weighted summation of the outputs of the neurons in the previous layer with synaptic weights as below

$$\begin{aligned} x_j(k) &= \sum_i w_{ji}(k-1) \cdot y_i(k-1) \\ y_j &= \varphi(x_j) \end{aligned} \tag{2.3}$$

where $x_j(k)$ is the state of j^{th} neuron in layer k, $w_{ji}(k-1)$ is the synaptic weight connecting between i^{th} neuron in layer k-1 to j^{th} neuron in layer k, and $\varphi()$ is the non-linear activation function such as sigmoid or hyperbolic tangent function [35] ($N.L()$ in Fig. 2.1 (a)) to generate the output of the neuron.

For the given input, the output of MLP is determined by feedforwarding from the input layer to output layer with all synaptic weights (inference). In other words, the operation of MLP is determined by synaptic weights by training. In training, all weights are randomly initialized and updated with training data and its desired output (supervised training). Based on the error ($E(n)$) between the output of MLP and desired output and its back-propagation, error sensitivity of each weight to network output (gradient: ∂E/∂w_{ji}) is computed to change synaptic weights.

Fig. 2.2 (b) illustrates that gradient is computed from output layer back to input layer to minimize the error at the output layer. According to the error sensitivity (gradient), each weight at iteration n+1 is updated as follows:

$$w_{ji}(n+1) = w_{ji}(n) - \eta \cdot \frac{\partial E(n)}{\partial w_{ji}(n)} \tag{2.4}$$

where η is learning rate and $E(n)$ is the error at iteration n. According to (2.4), synaptic weights are updated based on the gradient and the learning rate until the output error is less than predetermined threshold.

Although MLP inference with given pre-trained weight needs just feedforward phase (2.3) to compute the state of neurons at the output layer, training requires both feedforward phase (2.3) and feedback phase (2.4) and these two phases are iterated until error at the output layer is low; thus it demands huge number of computations.

2.1.2 Convolution Neural Network (CNN)

Compared to MLP, composed of only fully connected layer, convolution neural network (CNN) places multiple pairs of convolution layer and pooling layer before series of fully connected layer. Fig. 2.3 shows example of CNN with three 2D convolution layers, two max pooling layer, and two fully connected layers for scene labelling application [38].

Fig. 6.4 shows the operation of convolution in feedforward and backpropagation in detail. 2D convolution layer is sweeping a 2D kernel on the 2D image to generate the 2D

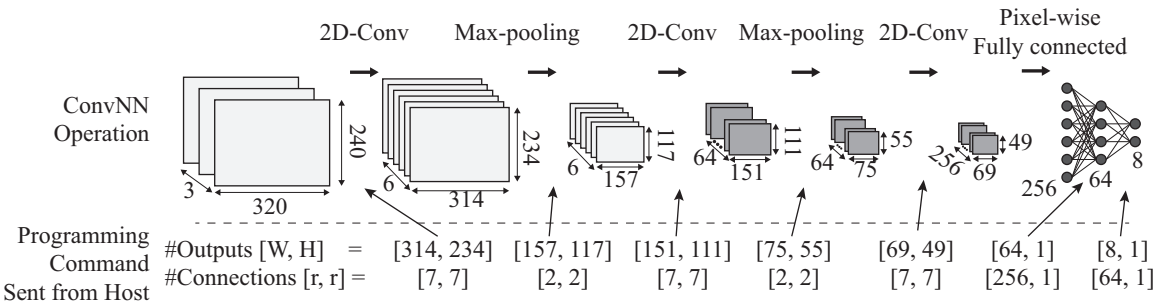


Figure 2.3: Convolution neural network composed of three 2D convolution layers, two max pooling layer, and two fully connected layers.

output. If input image has multiple channels (sometimes it is called as 'depth'), 3 channels for RGB image for example , the kernel has also same number of channels. Convolution is performed per each channel and its output is accumuated across the channel; therefore the channel of output is 1. For N_O kernels, the dimension of output is $N_I \times H_O \times W_O \times N_O$. The equation of convolution for $n_i^{th}X$ is described as below

$$\begin{aligned} h_i = h_o \times V_{stride} + h_k - H_K/2, \;\; w_i = w_o \times H_{stride} + w_k - W_K/2, \\ y(n_i, h_o, w_o, n_o) = \sum_{d_k} \sum_{h_k} \sum_{w_k} x(n_i, h_i, w_i, d_k) \times w(n_o, h_k, w_k, d_k) \end{aligned} \tag{2.5}$$

The main role of convolution layer is extracting spatial features in $R^{W_I H_I}$ from the input and the target feature is defined by a kernel. After convolution with N_O kernels, N_O outputs represent the existence of the feature on (h_o, w_o). Starting from random kernels, training CNN finds meaningful kernels to classify the object by itself. From 2012, DNN with convolution layers shows great accuracy in image recognition field [1].

After extracting features, pooling layer may be placed to reduce the size of input for the next layer. The most common ways in pooling is taking maximum value in the pooling radius or averaging the value in the pooling radius. After passing two pooling layers, the output image size is less than 5% of original input size in Fig. 2.3. At the end of extracing features, two layered MLP is placed (two fully connected layers) as a classifier. Compared to MLP, its input is not original image, but image indicating the location of feature (feature

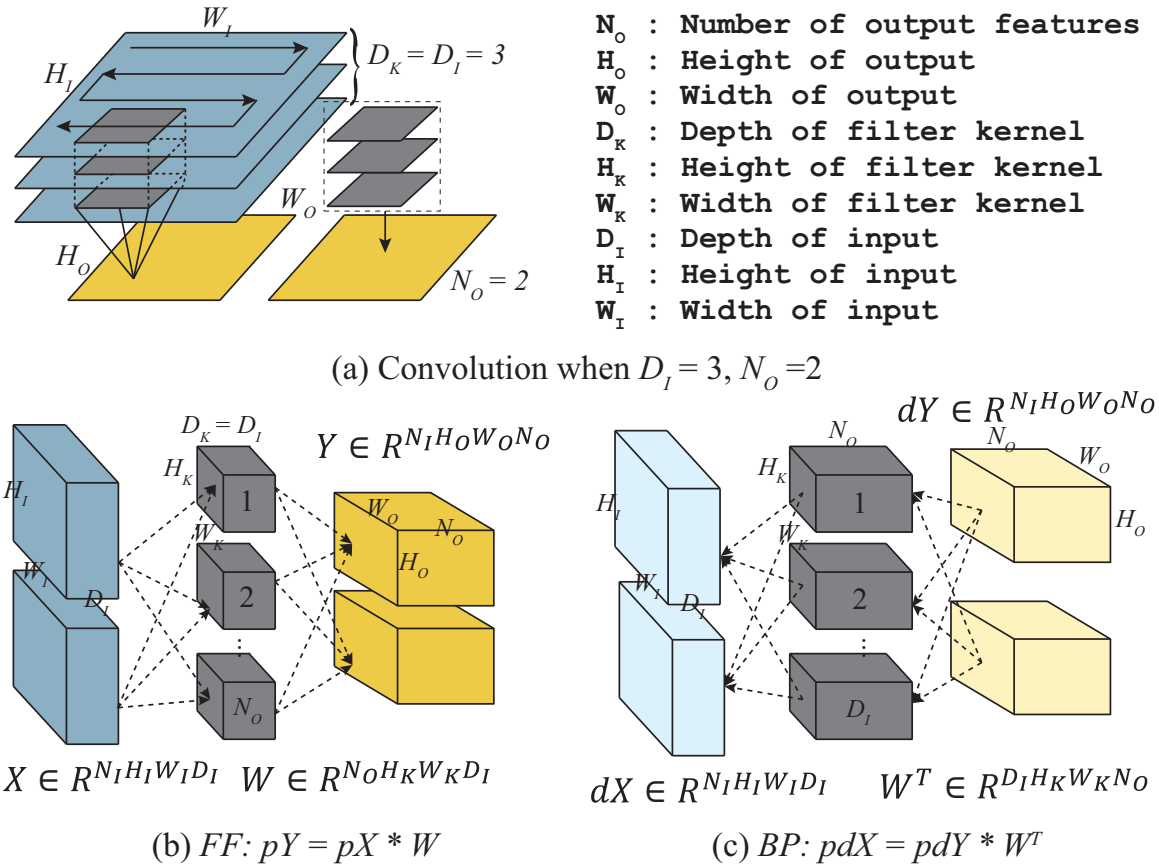


Figure 2.4: Operation of convolution layer in feedforward and backpropagation.

maps).

2.1.3 Recurrent Neural Network (RNN)

Both MLP and CNN are feedforward network, where the activation of neuron in layer i propagates to the next layer $(i+1)$ through weighted connections (either fully connected or spatial locally connected). After passing a single image $X[n]$ (input at time n), there is no output remaining in the network about $X(n)$ and all values in the network need to be re-generated for next input $X[n+1]$ (input at time $n+1$). In other words, inferencing $X[n+1]$ is independent with the inference result of $X[n]$ (time independent). It's impractical for analyzing time dependent data, such as speech, video, sentences.

To add *time* in deep neural network, recurrent neural network (RNN) was introduced to

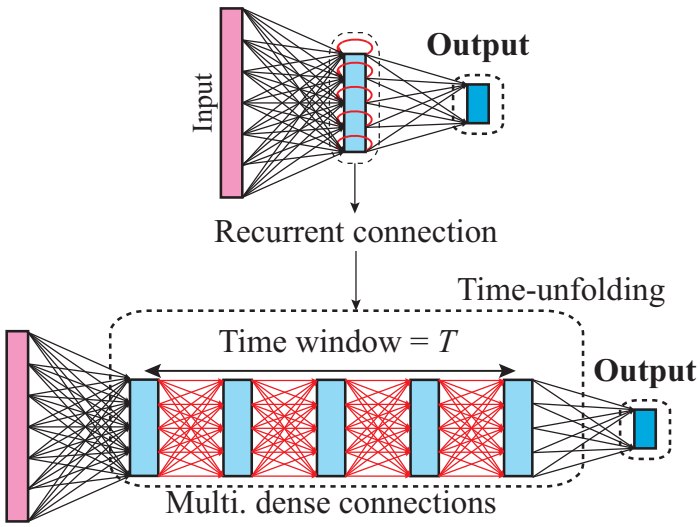


Figure 2.5: Recurrent neural network and its time unfolded network.

represent time information using time-unfoling [39]. Fig. 3.5 shows simple RNN structure and its time unfolded network. A single neuron in RNN takes all activation from both previous layer and current layer. In other words, the activation (y[n]) generated by $X[n]$ is feeded as input for computing activation of $X[n+1]$. The activation of the neuron in RNN is computed as

$$x_j(k) = \sum_i w_{ji}(k-1) \cdot y_i(k-1) + \sum_l w_{jl}(k) \cdot y_l(k). \tag{2.6}$$

For finite time window T, recurrent layer can be unfolded to T fully connected layers to store T previous activations ($y[n-1]...y[n-T]$).

Variations of RNN have been introduced (LSTM [39], GRU [40]) by adding additional layer besides to their hidden layer of RNN for improving training quality. Since additional layer is also fully connected layer, both variations can be considered as multiple stacked fully connected layer after time unfolding.

Similar to training with backpropagation in feedforward network, training RNN requires propagate the error from the output to the input, but traverse in time domain as well. Backpropagation Through Time (BPTT) is backpropagation of the RNN in time-unfolded network structures [41]. As the number of layers in time-unfolded RNN is time window

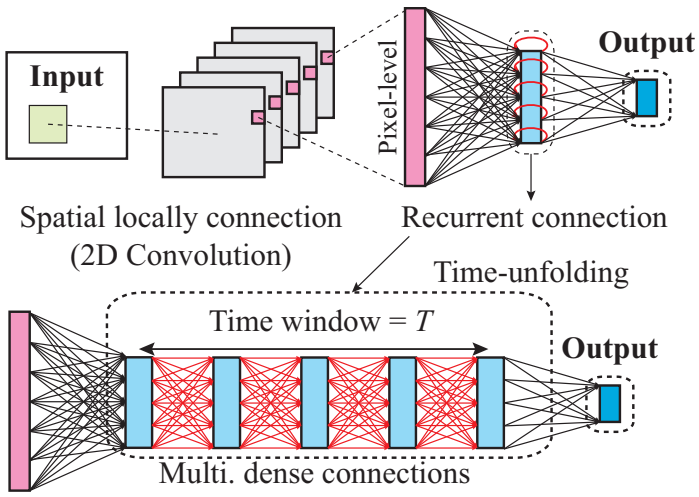


Figure 2.6: Simplified example of deep learning for generating sentence for image description [42]. It is composed of 2D convolutional layer, recurrent layer, and fully connected (dense) layer. Recurrent connection (red line) can be transformed multiple dense connections after time unfolding.

T, the training overhead in terms of latency or memory requirement is proportional to time window.

2.1.4 Deep Neural Network (DNN)

To approximate complex non-linear behavior of application, there have been extensive studies on more complicated network structures by adding different types of layers or connecting two non-adjacent layers [43] by direct paths. For example, convolution layers are placed in front of RNN to extract image features and generate sentences for image description [42] (Fig. 2.6). As I already explained, RNN becomes MLP with T layers and T becomes large to track long time windows.

Artificial neural network with many layers are called as deep neural network (DNN) and machine learning using DNN is called as deep learning (DL). Network with many layers becomes challenge in several ways.

First, overall memory capacity requirement increases significantly. Accelerator should store both neurons' activations and synaptice weights value between two neurons. For simple application, such as hand written number recognition (MNIST [9]) with three layered

MLP, 10MByte memory capacity is large enough to store all neurons and weights in single precision (32bit). However, recent DNN [43, 44, 45] shows that required memory capacity is about 500 Mbyte. As total memory capacity increases above on-chip cache memory capacity, deep learning architecture design should consider external DRAM in the system and interface between processing core and memory becomes the most important bottleneck in entire design.

Second, the number of operations also increases; therefore operating the accelerator for real time application becomes challenge. Image recognition developed by Google Inc. [44] requires 180 Giga-FLOP (GFLOP) to inference a single image. In other words, 5.4Tera-FLOP/second (TFLOPS) is required approximately to process 30 frame-per-second (fps). Training, which requires $\sim$3 times the number of operations of inference, becomes more challenge even with high performance GPU [46].

Third, applying error backpropagation for training becomes challenge due to gradient ($\partial E(n)/\partial w_{ji}(n)$) vanishing (or exploding) issue as gradient is passed through the N layers and multiplied with small (or large) values N times [35]. To overcome this challenge, LSTM [39] is introduced in RNN or ResNet [43] in image recognition application.

2.2 Training deep neural network with gradient descent

In this section, we will explain the approach for training DNN with gradient descent, which is composed of three steps: feedforward, backpropagation, and weight update in recent DNNs [35]. Fig. 2.7 shows a simple DNN and its feedforward, backpropagation, and weight updating for different types of the layer in the network.

2.2.1 Feedforward (FF)

Feedforward is propagation of neuron activation from i^{th} layer to $i + 1^{th}$ layer through weights between two layers. The output of a neuron (state) is weighted summation of activation from all connected neurons in previous layer (and current layer as well for

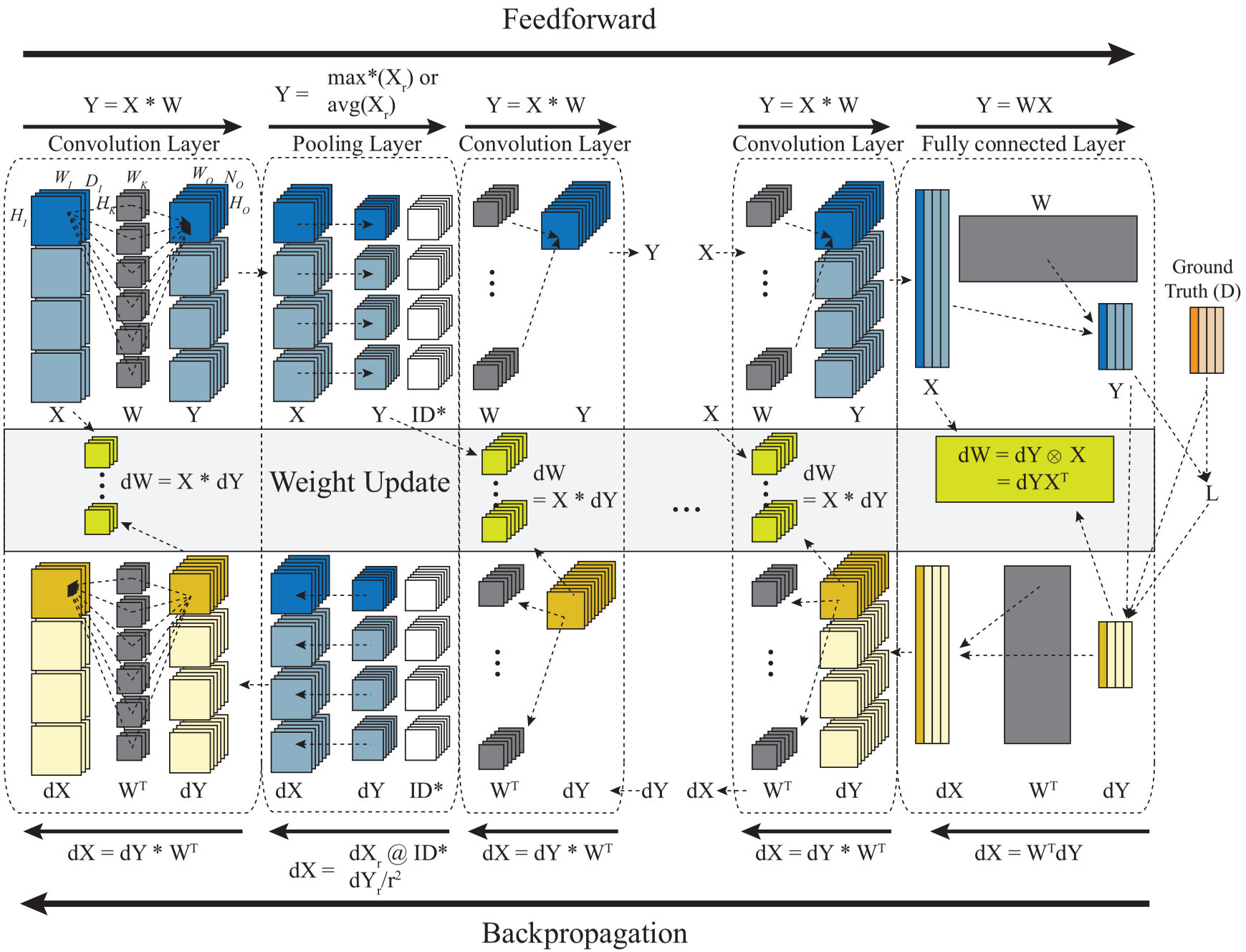


Figure 2.7: Deep neural network composed of convolution layer, pooling layer, and fully connected layer. It has three phases: feedforward, backpropagation, and weight update.

RNN [47]). It is the only phase required during the inference. Fig. 2.7 shows feedforward through convolution layer and fully connected layer.

2.2.2 Backpropagation (BP)

For a given input, at the end of feedforward operation, the output of the last layer is compared with the *ground truth* i.e. the desired output for this input and computing loss (L). The loss can be defined by simple mean squared error (MSE) or combination of softmax layer and cross-entropy layer [48]. Backpropagation is the phase to find the impact of each state on the loss (gradient) $\partial L/\partial X(dX)$ by propagating from the last layer. Since there is no definition of loss (L) in hidden layer, $\partial L/\partial X(dX)$ is computed from the dX of $i+1^{th}$layer ($\partial L/\partial Y(dY)$) instead of computing $\partial L/\partial X(dX)$ directly. We can see most of *arithmetic* operation in Backpropagation is similar to that of Feedforward in convolution and fully connected layer except *transposing kernel* (Fig. 2.7).

2.2.3 Weight Updates (UP)

Based on dX, $\partial L/\partial W(dW)$ needs to be generated to reduce L in next iteration (epoch). New W for next iteration is determined as $W_{new} = W_{old} - \eta \cdot dW$, where η is learning rate. Recently, additional term is added during the update such as momentum [10]. For convolution, it needs convolution between X and dY. As dimension of dY is smaller than X by radius of kernel (W), it is convolution with very large kernel size. For fully connected layer, it requires vector vector outer product. Thus we can see there is *additional* operation and data flow is needed for efficient operation in weight updating.

2.2.4 Data Preparation (Prep)

For each operation, input data need to be pre-loaded into the memory to improve data flow between memory and processing engines. If the layout of output generated in layer i does not match with required data layout for the layer $i+1$, it needs to be re-arranged between

multiple memory banks. In addition, for convolution, to make the size of the output same as the size of input, the input needs *dummy* zeros on its boundary.

2.2.5 Minibatch Training

Mini batch training involves updating weights after training small set (K) of training data. If total number of training data is N, it will iterate K/N times for one epoch. It is faster than training with large batch sizes and shows smoother convergence than training individual images. Moreover, it can reuse weights K times improving computing efficiency [35]. However, it requires more on-chip memory to store K temporal data.

The multiple mini-batches are trained in parallel using multiple computing nodes where each node independently compute $\partial L/\partial W(dW)$. After generating all dWs, all machine share updated new W (synchronous training) [49]. To overcome the unbalance in training latency among multiple nodes, in asynchronous training once a node generates dW, it will have new W while others use old W [50].

2.3 Deep Learning Accelerator

2.3.1 Architecture Based on On-Chip Memory

Early stage deep learning architecture design focuses on specific connection type, convolution layer or fully connected layer. As convolution was important operation in image processing domains, many convolution accelerator design were introduced [20, 21, 22, 23, 24, 25, 26, 27, 51, 52] in both FPGA and ASIC platform. Thanks to kernel sharing, all kernels can be stored in on-chip local memory and input is streamed from the outside. With high data reusage in both input (X) and kernels (W), data movement from the outside can be minimized.

To cover large network with many parameters, weights are pruned or compressed to be fitted into on-chip memory with small accuracy degradation [26] or eDRAM is used as local memory [24]. However, it is limited to specific pruned network [53] or network size

Table 2.1: 3D Stacked Memory Specification.

	DDR3 [54]	Wide I/O 2 [55]	HBM [56]
Interface	2D	3D	2.5D
Max. # Channels	2	8	8
Word Size	64 bit	128 bit	128 bit
Peak B.W.†	12.8 GBps	6.4 GBps	16 GBps
$t_{CL} + t_{RCD}$	25 ns	N/A	N/A
Operating Voltage	1.5 V, 1.35 V [54]	1.1 V [55]	1.2V [56]
Energy	70 pJ/bit [57]	N/A	N/A

†*Peak bandwidth per channel*

Table 2.2: Hybrid Memory Cube 1.0 Specification.

	HMC-Ext [58]	HMC-Int [58]
Interface	3D	3D
Max. # Channels	4	16
Word Size	32 bit	32 bit
Peak B.W.†	40 GBps	10 GBps
$t_{CL} + t_{RCD}$	27.5 ns [59]	27.5 ns [59]
Operating Voltage	1.2 V [60]	1.2 V [60]
Energy	10 pJ/bit [60]	3.7 pJ/bit [60]

†*Peak bandwidth per channel*

less than 100 MBytes [24].

In conclusion, previous architecture designs based on on-chip memory lack of scalability due to memory capacity limitation and lack of programmability since it is designed to specific operations.

2.3.2 Architecture Based on Processor-in-Memory

High density 3D memory, composed of multiple stacked DRAM dies offers to meet the capacity and bandwidth demands of neuro-inspired computations. Table 2.1 compares several candidate 3D memory technologies.

Wide I/O-2 is designed for mobile platforms by stacking conventional DRAM on the mobile SoC (3D interface) [55]. Using high density TSVs, the number of I/Os per channel is high (total number of I/Os from 8 channels is 1,024). High Bandwidth Memory (HBM)

is designed for high performance processors [61]. HBM is composed of 4 DRAM dies and one single logic die. The logic die is designed for testing (design for test (DFT)), TSV arrays, and interface (PHY) for communication with the SoC.

The Hybrid Memory Cube (HMC) is also designed for high performance applications [58]. It is composed of multiple stacked DRAM dies and a single base logic die interconnected with TSVs. Each DRAM die is divided into 16 partitions in a 2D grid and the corresponding partitions on the vertical die form a single *vault*. Each vault has an independent vault controller on the logic die; therefore multiple partitions in the DRAM die can be accessed simultaneously. There have been proposals for off-loading data-intensive operations onto the logic die [62, 63, 64].

Compared to HBM or Wide I/O-2, the HMC architecture provides highly parallel access to the memory (one channel per vault) which is better suited to the highly parallel architecture of the computing layer in the Neurocube. The logic and memory dies can be fabricated in different process technologies; e.g., DRAM dies are fabricated in 50nm and logic die is fabricated in 28nm [60]. However, the area of the logic die relative to the memory dies is constrained by the package [60], and power dissipation is limited by the much tighter thermal constraints [65]. In this thesis, I will refer to the interface between DRAM layers and the logic die as 'HMC-Int' (3^{rd} column in Table 2.2).

As HMC allows flexibility on the logic die design, placing the processing elements below 3D stacked DRAMs and communicating through TSVs, processor in memory (PIM) architecture, has been investigated in terms of architecture design [28, 29, 30, 62, 66, 67], thermal analysis [65, 67, 68], and package level [60, 67] to leverage memory interface overhead.

Especially for deep learning accelerator, PIM architecture based on HMC shows great potential because of high memory bandwidth by multiple parallel memory channels and simple processor design (mainly fixed point multiply-accumulator unit) [28, 29, 30]. In this thesis, proposed architecture design is based on PIM architecture with HMC 1.0 since

it is only HMC version reported in silicon level in terms of power and footprint.

2.3.3 Approximate computing in Deep Learning Accelerator

Many studies have been done to mathematically analyze error sensitivity of neural network's output due to small perturbation (uniform distribution or normal distribution) at the inputs and/or the weights [69, 70, 71, 72]. An algorithm extending layers sensitivity to next layer from input layer to output layer has been proposed [69]. However, the impact of error during training on the inference was not explained. In [70], the sensitivity of neural network among different sets of weight matrices is discussed.

More interestingly, the effects of analog noise (small perturbation) in synaptic weights during training on fault tolerance and performance of network are studied [71]. This work provided mathematical fundamentals and simulation results showing that small perturbation on synaptic weights during the training can improve fault tolerance of MLP for inference under the error in synaptic weight or input and accelerate training of MLP i.e., cost function of MLP converges fast during the training (less iteration). But too much error significantly increases training time and cannot guarantee the accuracy of target operation. The small perturbation, however, is limited for only uniform distribution; therefore this error distribution is not practical for quantization error caused by limited bit precision. The impact of bit-precision on the number of iterations for XOR training (small perturbation) is also discussed in [73], showing similar results as in [71].

Recently, approximate computing by reducing bit precision [72, 74, 75, 76] or using inexact multipliers [72, 75, 77, 78] in neural network have been proposed. The closest work is the one presented in [74] where some LSBs are forced to zero in selected neuron's state to reduce power dissipation [74] of MAC.

An inexact multiplier, simplifying design by allowing bit error on some LSBs, is designed with iterative logarithmic multiplier having computation error less than 1%. Based on the probabilistic logic minimization algorithm, different types of inexact multipliers

were designed and applied for simple MLP network [78]. Inexact multiplier can reduce both power consumption (x2.67) and area overhead (x1.46). In [78], only small MLP network was studied and all neurons and weights are approximated.

CHAPTER 3

IMPACT OF QUANTIZATION DUE FO FIXED POINT ON DEEP LEARNING

Both NeuroCube and NeuroCubeuse a fixed numeric representation (16bit fixed point) to represent weights and neurons' activation. Although general purpose unit such as GPU or CPU adopt *floating point* to avoid overflow and achieve high resolution, most of previous DNN accelerators use *fixed point* to save area and power overhead with small accuracy degradation [20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 33, 51, 52].

For given fixed point multiplier or adder (ex: 32bit) after accelerator design, dynamic power consumption is proportional to *the number of actual toggling bits*. Fig. 3.1 shows the dynamic power consumption of 32bit fixed point multiplier designed in 130nm operating in 200MHz. Although there were some papers illustrated MLP hardware with 8-bit synaptic weights [79],[80], these papers do not consider on-chip training. As I will explain later, for on-chip training, digital DL hardware needs higher precision due to gradient. Moreover, as optimal bit precision for both inference and training strongly depends on benchmark or network structure, I assume 32-bit fixed point for both synaptic weights and states as a base line to avoid loss of generality.

Fig. 3.1 shows the power can be saved by 53% by reducing the bit width from 32 bits to 8 bits. The error of multiplier for given precision is also simulated by multiplying two randomly generated operands and comparing with 32-bit reference. Fig. 3.2 (a) shows the cumulative error distribution for 16-bit fixed point ($Q_{1,7,8}$: 1 sign bit, 7 integer bits, and 8 fractional bits). The probability of quantization error increases as bit width decreases from 32 bits and decreasing to 12 bits and 8 bits have quiet large error due to overflow.

Another approach to save the power in DL accelerator is applying an inexact multiplier. An inexact multiplier, simplifying design by allowing bit error on some LSBs, is designed with iterative logarithmic multiplier having computation error less than 1%. Based on

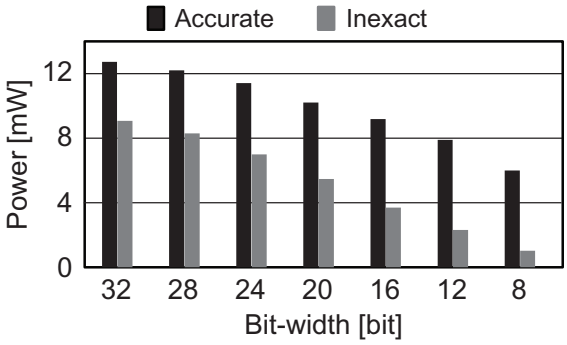


Figure 3.1: The power dissipation of the accurate multiplier (black) and the approximate multiplier (gray).

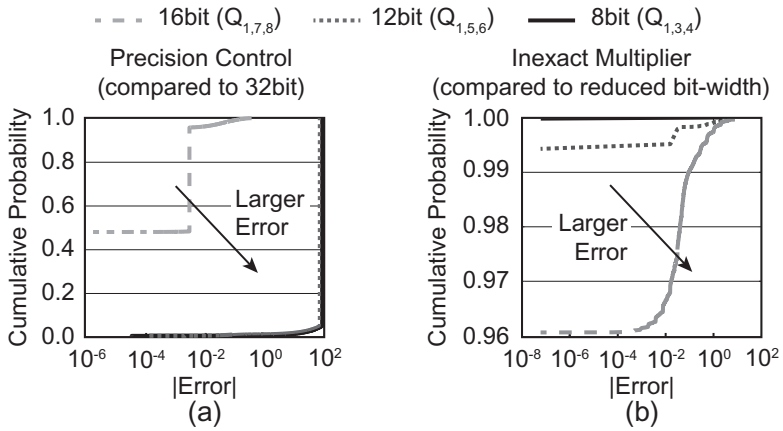


Figure 3.2: The cumulative probability of (a) quantization error due to precision control and (b) the additional error induced by using an inexact multiplier with error correction of 20 MSBs (x-axis is logarithmic scale).

the probabilistic logic minimization algorithm, different types of inexact multipliers were designed and applied for simple MLP network [78]. Inexact multiplier can reduce both power consumption (x2.67) and area overhead (x1.46).

In the same manner, cumulative probability of error due to inexact multiplier with 20-bit recovery (20 MSBs will not have error) for each bit precision is simulated (Fig. 3.2 (b)). As there is no error in 20 MSBs, more than 95% of errors are smaller than 10^{-2}. Interestingly, the additional error due to inexact multiplier decreases in reduced bit precision since some of LSBs are forced zero. Therefore, errors from inexact multiplier are hidden. It allows more aggressive power saving with very small additional error.

For the given DL accelerator, how many bit precision is needed or how many weights or activations could be approximated using the inexact multiplier can vary for different deep learning application. There is no *golden number* of bit precision for all DL applications, however, Kung et al. suggested allocating bit precisions for weights based on error sensitivity (gradient), generated during the back propagation under accuracy degradation budget [72].

According to [72], after designing the accelerator using both exact multiplers and inexact multipliers, assigning bit precision and multiplier types (exact vs inexact) can be pre-determined before inference operation based on training result.

In this chapter, I focus on the dense network (such as MLP or RNN) since the impact of approximating weights on the power consumption is large in dense connection rather than locally sparse connection (convolution layer).

3.1 Greedy Algorithm to Choose Near-Optimal Bit Precision for Low-Power Design

In this section, I will briefly introduce greedy algorithm to choose near optimal bit precision for synaptic weights under given hardware adopting both exact and inexact multiplier [72].

Under given finite bit precision sets (for example, 32bit, 16bit, 8bit), fine-grained greedy algorithm was introduced (Algorithm 1) [72] to determine near-optimal bit precision for all synaptic weights, which minimizing power under the target accuracy. Starting from the lowest precision mode (lowest accuracy and lowest power consumption), it generates two different bit precision ratio sets (R_{trial1}, R_{trial2}) to improve accuracy. After feedforward with two candidates, it selects the set with high score (the weighted sum of normalized quality increase and the negative of normalized power increase). It iterates until it meets target accuracy. Detail explanation for Algorithm 1 is illustrated in [72, 75].

After R_{prec} is decided, type of MAC (accurate or inexact) is assigned for all synaptic weights based on sorted error sensitivity (Fig. 3.3 (a)). Fig. 3.3 (b) shows the trace of Algorithm 1 with three bit-precision modes: 32bit, 16bit, and 8bit with 0.9 target accuracy

Algorithm 1 Power-aware feedforward NN design methodology

Input:

Computed error sensitivity: Grad,
Estimated power: P_{est}
Ratio of approximate PEs: γ
Quality constraint: Q_{const}
Test data set D_{test}

Output:

Minimum power: P_{min}
Precision ratio set: R_{prec}

Initialize $R_{prec} \leftarrow \{1.0, 0.0, ..., 0.0\}$
[Q, P] $\leftarrow$ approx.ff($\gamma, R_{prec}, D_{test}, P_{est}$)
while $Q < Q_{const}$ **do**
  $[R_{trial1}, R_{trial2}] \leftarrow$ generate.next.Rprec(R_{prec})
  $[Q_{trial1}, P_{trial1}] \leftarrow$ approx.ff($\gamma, R_{trial1}, D_{test}, P_{est}$)
  $[Q_{trial2}, P_{trial2}] \leftarrow$ approx.ff($\gamma, R_{trial2}, D_{test}, P_{est}$)
  score1 $\leftarrow$ compute.score($Q_{trial1}, P_{trial1}, Q, P$)
  score2 $\leftarrow$ compute.score($Q_{trial2}, P_{trial2}, Q, P$)
  if score1 $>$ score2 **then**
    $[R_{prec}, Q, P] \leftarrow$ update($R_{trial1}, Q_{trial1}, P_{trial1}$)
  else
    $[R_{prec}, Q, P] \leftarrow$ update($R_{trial2}, Q_{trial2}, P_{trial2}$)
  end if
end while
$P_{min} \leftarrow P$
return P_{min}, R_{prec}

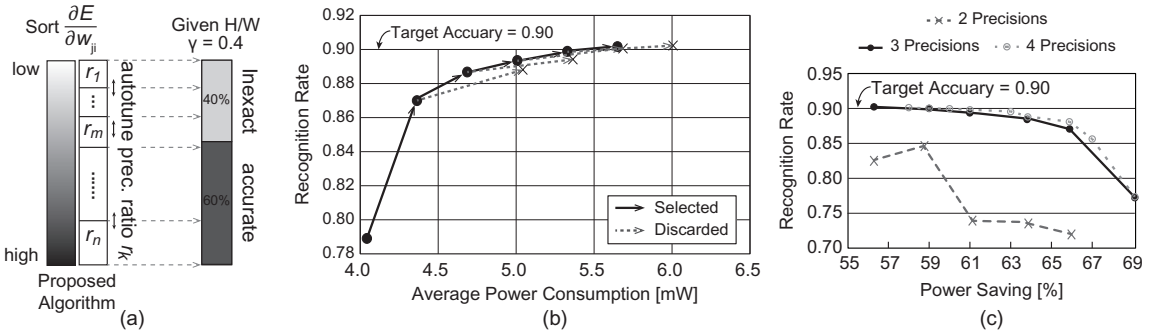


Figure 3.3: (a) Overview of the proposed greedy algorithm with =0.4. (b) Experimental result on quality-aware low-power design methodology. This method dynamically selects (solid line) precision bit-widths which increase accuracy with less power increase. (c) Recognition rate comparison between two, three, and four different bit-precisions at varying power constraints.

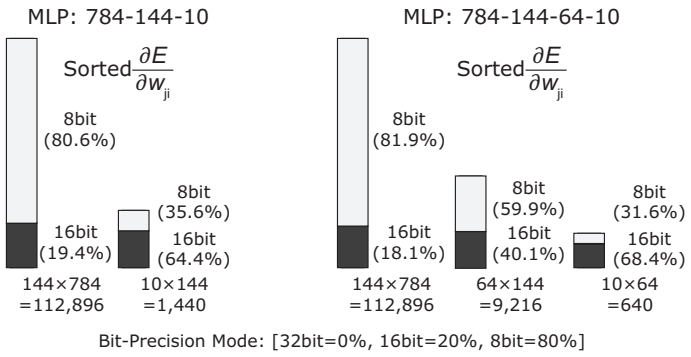


Figure 3.4: Analysis the ratio of approximate for each layer in MNIST with 3 layers(784-144-10) and with 4 layers (784-144-64-10).

for handwritten digit recognition (MNIST) [9] application with 3 layer MLP. As a result, 0.9017 recognition rate is achieved with 56% power saving compared to the result with all 32bit using 100% accurate PEs (0.9053 recognition rate is achieved). Although four different bit-precisions (32bit, 16bit, 12bit, and 8bit) are tried, the result is similar to the case with three bit precisions (Fig. 3.3 (c)). However, I can see two bit precision modes are not enough to save power of MLP with MNIST application.

3.1.1 Layer-by-Layer Distribution of Approximate Synapses

Algorithm 1 sorts all synaptic weights of the neural network regardless of layer. Therefore, each layer has different percentages of approximation under given total bit-precision mode (Ex: 32 bits: 0%, 16 bits: 20%, 8 bits: 80% for entire network). Fig. 3.4 shows the percentages of each bit precision for each layer to understand which layer allows more approximation.

For example, in MLP with 3 layers, first synaptic connections (between first and second layer) has 80% of 8 bits and 20% of 16 bits while second synaptic connections (between second and last layer) has 36% of 8 bits and 64% of 16 bits. In other words, synaptic connections between the earlier layers can be approximated more. The same observation is also made by increasing the number of layers in the MLP (see the result for 4 layers MLP in Fig. 3.4). It is also observed in [80]. The analysis suggests that future work may consider

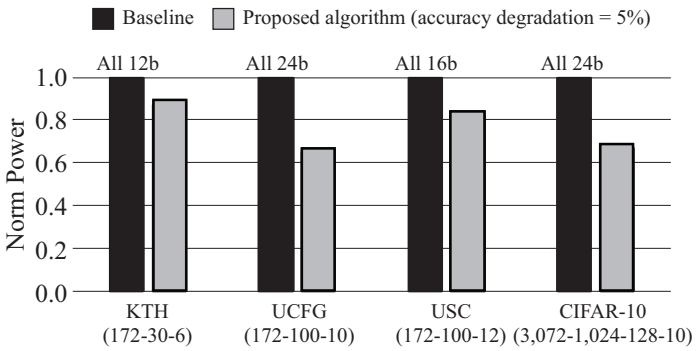


Figure 3.5: Proposed algorithm applied in complex network: RNN for human activity recognition [81, 82, 83] and MLP for CIFAR-10 [84].

using different sensitivity for different layer, layer-wise approximate should be studied in the future.

3.1.2 Proposed Algorithm in Complex Network

In this section, I tried proposed algorithm in more complex network: recurrent neural network (RNN) for human activity recognition [85] and MLP for CIFAR-10 [84] to study validation of proposed algorithm in complex network. MLP network for CIFAR-10 is 3,072-1,024-128-10, which is bigger than MLP for MNIST.

Although the number of hidden layers and neurons in hidden layer are low ($\sim$100 in this chapter), its recurrent connection can be transformed to T fully connected layers after time unfolding (inference based on T history); in other words, it is very deep T layered network (I set T as 50). Three different human activity recognition video dataset (KTH [81], UCFG [82], and USC [83]) are trained using backpropagation. RNN with a single hidden layer is used.

After training network, proposed algorithm is applied with three bit precision modes (16bit, 12bit, and 8bit). Fig. 3.5 shows the proposed algorithm can save power $14\% \sim 36\%$ while accuracy is degraded less than 5% compared to the baseline. Baseline for each benchmark is illustrated in the Fig. 3.5. I can see that assigning different bit precision for each synaptic weights based on gradient can save more power while maintain the baseline accu-

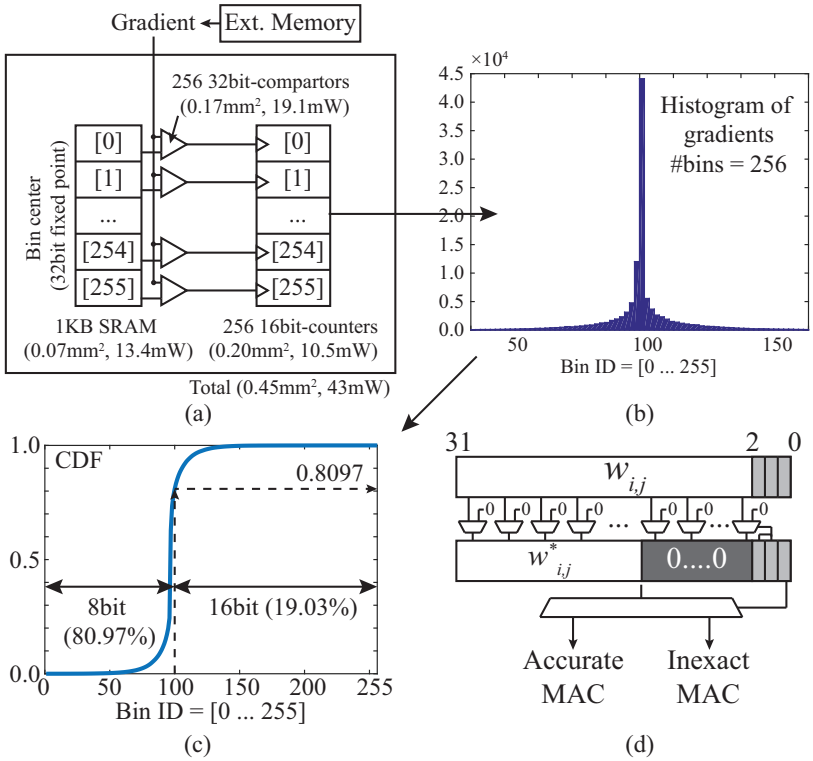


Figure 3.6: Binning gradients for proposed algorithm in on-chip. (a) On-chip implementation of proposed algorithm to generate histogram, (b) histogram of gradients with 256 bins, (c) assign first 101 Bins for 8bits and rest of them for 16bits, and (d) controller for approximation during the inference.

racy even considering additional power overhead of proposed algorithm (Section 3.1.3).

It shows that proposed algorithm can be also applied to complex network such as big MLP or RNN, very deep network trained using backpropagation and it can save power with less accuracy degradation.

3.1.3 Overhead of proposed algorithm in On-chip Level

For proposed algorithm in on-chip level, two hardware modules are additionally required. First, weights are sorted based on the gradients and grouped for given bit precision sets. After grouping, I need to force some of LSBs of weight and assign MAC unit types (accurate or inexact).

Grouping synaptic weights

The most important module for the hardware implementation of the proposed algorithm is the sorting engine where input data is the gradient stream from memory after training. The design of the sorting engine needs to consider the trade-off between power and performance. A simple and low-power approach is to design a Bubble Sorter using a single 32bit fixed point comparator. A 32bit comparator operating at 800MHz in 130nm CMOS requires $675\mu m^2$ of area and $74.5\mu W$ of power, incurring negligible overhead. However, the worst-case complexity of bubble sort is $O(n^2)$, therefore the latency for sorting all gradients of MNIST (784-144-10) is 16.4 seconds with 800MHz bubble sorter. Hence, the worst-case latency is significantly high. The latency can be improved using complex parallel sorting engine, such as a Radix sort engine [86]. I can estimate area/power overhead of Radix Sorter in 130nm process following prior work [86]. For example, a 250MHz Radix Sorter using 130nm require $2.6mm^2$ of area and 730mW of power incurring significant overhead. However, as the complexity of Radix Sort is linear to number of elements, total latency for sorting all gradients of MNIST (784-144-10) is estimated as 0.2mS.

Considering the latency-power trade-off associated with the sorting engine, I propose an alternative approach for the hardware implementation. I observe that the proposed algorithm only needs to group the weights based on the gradients, it does not necessary require sorting of the weights. Therefore, I propose to perform on-chip binning of the weights, rather than sorting. After training, based on given number of bins (ex: 256), I compute the bin centers and stored in SRAM (32 bit $\times$ 256 = 1KB) to generate histogram of gradients. A gradient from external memory is compared with 256 bin centers in parallel using 256 32bit-comparators. It triggers one of 256 16bit-counters to generate histogram. After comparing all gradients, histogram of gradients is transformed to Cumulative Distribution Function of the gradients. Based on ratio of precision, I can find set of bins for each precision mode. This approach is illustrated in Fig. 3.6. A binning approach may not generate exact percentages, however, the error introduced by binning is negligible. For

example, in the MNIST (784-144-10) application if I want 20% and 80% weights in 16bit and 8bit, respectively, the binning will assign 19.03% and of weights to 16bit and 80.97% of weights to 8bit. The proposed approach requires additional on-chip memory to generate and store CDF, but with a linear complexity ($O(n)$). Considering 800MHz SRAM, the latency required for the proposed approach for the MNIST application is 0.2mS, which is negligible compared to training latency (2,000 epochs: 17.9mS). The additional hardware (256 comparators, 1KB SRAM, 256 16bit counters) require 0.45mm^2 area (3% overhead) and 43mW or power (2% overhead). The energy consumption of the algorithm (8.6uJ) is also negligible compared to the training energy (74.47mJ for 2,000 epochs).

Controlling approximation during the inference

After grouping, last ($log_2(N) + 1$) bits of each synaptic weights, are assigned to represent approximation flag: $log_2(N)$ bits to represent N bit-precision modes and 1 bit to mark whether it uses in-exact MAC or not (result of Algorithm 1). In the example (3 bit-precision modes: 32bit, 16bit, and 8bit), last 3 bits for each synaptic weights are assigned for approximation flag. Therefor, there is no additional overhead for storing 'approximation information'.

Based on approximation flag, approximation controller in each PE checks last 3 LSBs of a synaptic weight, then forces some of LSBs as zero using 32 1bit 2-to-1 MUXs and delivers the synaptic weight to accurate MAC or inexact MAC unit (32bit 2-to-1 MUX). Due to its simplicity, its area (2%) and power (4%) overheads are negligible compared to 32bit fixed point MAC units.

Total hardware overhead for proposed algorithm is illustrated in Table 3.1. Compared to base architecture, hardware overhead is 7% in area and 3% in power.

Table 3.1: Hardware Overhead for Proposed Algorithm (in 130nm process)

	Area (mm^2)	Power (W)
256 Comparators†	0.173	0.019
1KB SRAM†	0.070	0.013
256 16b Counters†	0.202	0.011
Approx. Ctrl.	0.580	0.024
Total	**1.02 (7%)**	**0.07 (3%)**

† *On-chip implementation of proposed algorithm (Section 3.1.3)*

3.2 Interaction Between Training Conditions and Approximation During Inference

In Section 3.1, I trained the network using 64 bits floating point without any approximation (off-chip training) using MATLAB. By storing look-up table for both non-linear activation function and its derivative, digital deep learning accelerator can train the network as well (on-chip training). As approximation is based on the result of training (error sensitivity), Section 3.2 studies the impact of on-chip training conditions (bit precision, max. number of iterations, and number of layers in network) on the approximated inference.

3.2.1 Different Bit Precision during the Training

Analysis on the iterations and accuracy of training

Instead of 64-bit floating point using MATLAB with PC (off-chip training), fixed point with different bit precisions (on-chip training) are used for training MNIST [87] and CNAE-9 [88] (classifying the business into 9 categories based on the text description). However, error during the training can change not only accuracy but also number of iterations due to failure of convergence. Since number of iterations during the training is critical for both on-chip training and off-chip training, the impact of quantization on the number of iterations should be studied.

Fig. 3.7 shows the number of iterations and accuracy of MNIST and CNAE-9 for different bit precision of training. In this experiment, 4 different bit precisions were tried for

Table 3.2: Energy Analysis for training and inference based on with different bit precisions

Mode	Operation	Total Ops	Latency (sec)	Iteration	Total Sec	Total Power (W)	Total Energy (J)
1	Train @ 32bit	343,008	8.93E-06	2,211	1.97E-02	3.78	7.46E-02
	Inference @ 10% 16bit + 90% 8bit	114,336	2.98E-06	1	2.98E-06	2.15	6.41E-06
2	Train @ 24bit	343,008	8.93E-06	3,561	3.18E-02	3.52	1.12E-01
	Inference @ 100% 8bit	114,336	2.98E-06	1	2.98E-06	2.09	6.22E-06

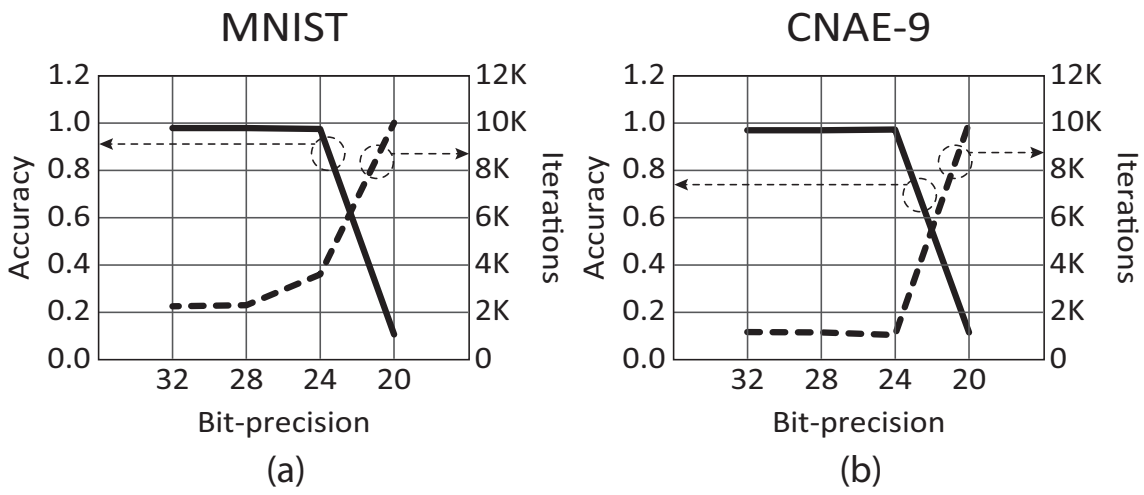


Figure 3.7: Number of iterations and accuracy for MNIST (a) and CNAE-9 (b) with different bit precision in training.

training: $Q_{1,7,24}$ (32-bit), $Q_{1,7,20}$ (28-bit), $Q_{1,7,16}$ (24-bit), and $Q_{1,7,12}$ (20-bit). For inference (classification), 32-bit fixed point is used for all cases. Iteration during the training is stopped when root mean square error computed at the output layer is below the threshold for each benchmark. The maximum number of iterations is limited to 10,000.

According to Fig. 3.7, I should note that higher bit precision is required for training compared to inference; I need at least 24 bits for training while I can use 16 bits for inference. Main reason for high bit precision in training is gradient [89]. For example, during the training of MNIST with 64 bits floating point, minimum magnitude (resolution) for gradient ranges $2^{-20} \sim 1^{-20}$.

According to [71], small perturbation due to limited precision may reduce the number of iteration of training; it is observed for CNAE-9; 24-bit and 28-bit precision has less iterations than 32 bits. For the 20 bits, however, number of iterations increases dramatically as the amount of error increases. For MNIST, the statement from [71] that small perturbation during the training decreases the number of iterations does not follow. First of all, small perturbation in [71] is uniform distribution while quantization error for 24 bits is always positive since LSB are forced to zero and its maximum value is around

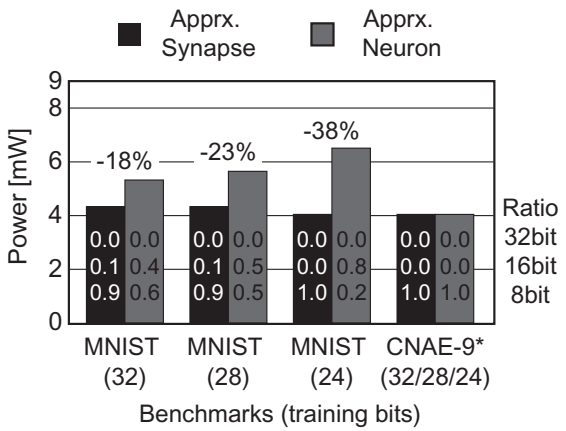


Figure 3.8: Comparing power consumption for different benchmarks (MNIST and CNAE-9) between approximating synapses and approximating neurons. Three bit precisions are tried: 32 bits, 28 bits, and 24 bits.

($1.5 \times 10^{-5} \approx 2^{-17} + \cdots + 2^{-21}$, since last 3 LSBs are approximation flag). As quantization error is always positive, the impact of accumulated error is different with that of accumulated uniform distributed error (cancel out each other in the accumulation) [71]. Moreover, the quantization error exists in every operands (states, weights, and gradients) while small perturbation exists in only weights [71].

It emphasizes that selecting appropriate bit-precision is very critical for training quality (accuracy) and energy efficiency in training (power consumption and training iterations). Moreover, the impact of different bit precision during the training on the approximation in inference will should be studied to maximize power saving (Section 3.2.1).

Comparing approximate neurons and synapses for different training bit precision

Approximate synapses [75] and neurons [74] are compared based on the different bit precisions in training (Fig. 3.8). Based on different bit precisions in training, error sensitivities are pre-computed and applied to the proposed algorithm to approximate neuron or synapse. In both approximation approaches, the number of approximated synapses are same. Accuracy degradation budget is set as 8%; 8% accuracy degradation from inference with 32-bit fixed point precision ($Q_{1,7,24}$) is allowed for each training conditions.

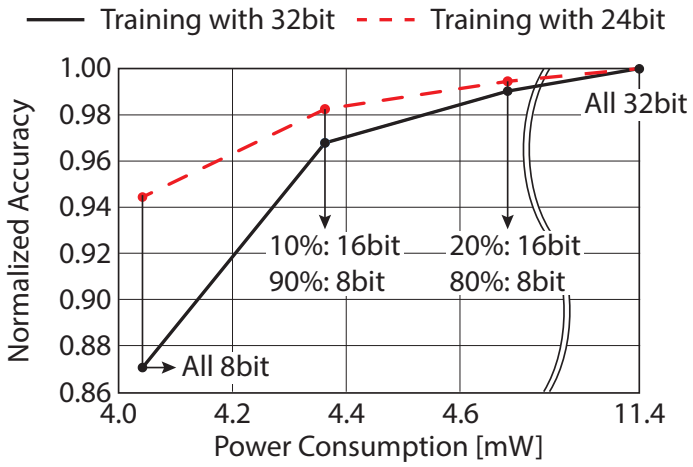


Figure 3.9: Comparing MNIST trained with 32bit and trained with 24bit in terms of power and accuracy.

First of all, CNAE-9 is simple enough to be trained with 8-bit fixed point for either synapses or neurons; therefore, there is no difference between approximate neurons and synapses. In contrast, for MNIST, approximate synapse can have more 8-bit precision for all training conditions. Therefore, approximate synapse can save more power than approximate neuron for all training conditions. Interestingly, for MNIST, approximate synapse can have more 8-bit precision ratio when MNIST is trained with low precision. It means that training with lower bit (with small quantization error) can make MLP more robust to small perturbation during the inference; it can approximate more synapses. For approximate neurons, however, this trend is not observable. Training with less bit precision has less chance to approximate neurons during the inference. As the error due to approximate neuron is not small perturbation (it may approximate some synapses, which are highly sensitive to the error at the output), MLP trained with less bit cannot overcome the error from the approximate neuron (more accuracy degradation). The relationship between bit precision in training and approximation in inference will be covered in next section.

Impact of different bit precisions in training on power saving in inference

For the rest of this chapter, I will use MNIST as example to see the relationship between training conditions and approximation in inference since CNAE-9 can be trained with even 8-bit precision.

MNIST trained with 24 bits and trained with 32 bits are compared in detail (Fig. 3.9). In the proposed algorithm, target accuracy is swept from 1.0 to 0.8. MNIST trained with 28 bits shows exact same results with MNIST trained with 32 bits. From this simulation, I can see MNIST inference does not need 32 bits; 16-bit synapses (20%) and 8-bit synapses (80%) shows less than 1% accuracy degradation for both 24 bits trained and 32 bits trained. If the target accuracy is 0.90, 32 bits trained MLP needs 16bit for 10% of synapses while 24bit trained MLP can operate with 8 bits for all synapses, which shows the similar result from [71]. Therefore, MLP trained with 24 bits can approximate more synaptic weights with low precision (more power saving) than MLP trained with 32bit.

For MNIST, however, 24 bits operation increases the number of iterations during the training (Fig. 3.7 (a)) about 60%. Therefore, based on the use-case of application (training once or training frequently), designer should consider proper bit precision due to the trade-off between training time and power saving during the inference.

Since bit precision for training changes not only training quality but also number of iterations (Fig. 3.7), its energy overhead should be compared with energy saving during the inference. Latency for training one iteration is computed using $38.4GOPs/s \times N_{Ops}$ (number of operations) and N_{Ops} for training fully connected layer from M neurons to N neurons are defined as $3 \times M \times N$ [90]. Table. 3.2 shows two modes comparison from Fig. 3.9: (1) 32bits training and 10% 16bits + 90% 8bits inference and (2) 24bits training and all 8bits inferences to achieve 0.94 accuracy using the hardware illustrated in [72, 75]. By reducing bit precision during the training, 24 bit training requires more energy (+37.4 mJ). During the inference, however, it can save $0.19\mu J$. In other words, if this system will MNIST inference more than 200,000 times, 24 bit training is beneficial, otherwise, 32 bits

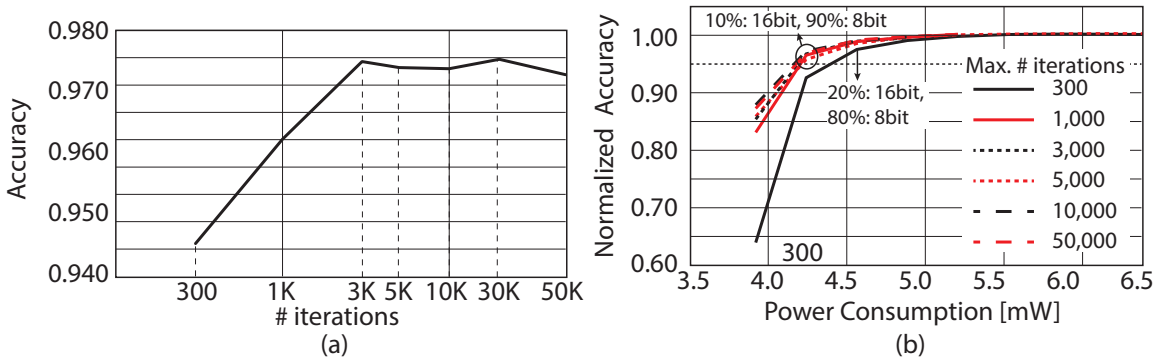


Figure 3.10: Impact of number of iterations in training of MNIST. (a) MNIST accuracy for different number of iterations. (b) Normalized accuracy and power saving using proposed algorithm based on different training iterations.

training is beneficial in terms of energy.

In conclusion, approximating synapse can save more power than approximating neuron during the inference under the given accuracy degradation. Moreover, power saving with approximated synapses increases when bit precision during the training is reduced carefully. However, training with low bit precision may increase the number of iterations. Therefore, based on the use-case, bit precision for training should be considered.

3.2.2 Different Number of Iterations during the Training

One of the difficulties during training is selecting a maximum number of iterations as it strongly depends on training data set, randomly initialized weights, and learning rate. In this section, I analyze the impact of maximum iterations during the training on the inference with approximate synapses.

For MNIST, maximum iterations are varied from 300 and 50,000. For all cases, 3-layer MLP network (784-144-10) with 32 bits is trained. As the number of iterations for MNIST training with 32 bit fixed point is about 2,000 in average (Fig. 3.7 (a)), training with 300$\sim$1,000 iterations shows lower accuracy (0.9460 and 0.9651) compared to training with iterations up to 3,000 (0.9793). Fig. 3.10 (a) shows the relationship between accuracy and maximum number of iterations in training. After 3,000, I can see there is no improvement

on the accuracy (fluctuation less than 0.5%).

Since the baseline accuracies (inference with 32bit fixed point without any approximation) are different for all cases, Fig. 3.10 compares the power saving and its normalized accuracy based on the baseline accuracy. Training with less iterations (300 or 1,000) shows more accuracy degradation for the same power saving. In other words, under the given accuracy degradation budget, increasing the number of iterations during training can save more power during inference by allowing more approximation on the weights. For example, assumed that maximum allowable accuracy degradation is 5%. Consider two training conditions, one with 3,000 iterations and the other with 300 iterations. The inference based on training with 3,000 iterations, can use 16 bits for 10% synapses and 8 bits for 90% synapses. On the other hand, when 300 iterations are concerned 16bit is used for 20% synapses and 8bit for 80% synapses. For the example in Fig. 3.10, I did not observe much difference in power saving for more than 3,000 iterations during training.

In summary, although training with less iteration could save training energy with negligible accuracy degradation, less-trained network is highly sensitive to approximation during the inference. In other words, training with enough number of iterations provides more potential for power saving using approximate synapse during the inference. This is an important observation especially for applications where training is performed rarely but inference is performed frequently. For such cases, a performance loss (longer) during training can lead to more power saving during inference.

3.2.3 Different Network Structure (Different Number of Layers)

The number of hidden layers and number of neurons in those layers are important design parameters for a MLP (or any deep learning network). In general, network with more layers is believed to be able to realize more complex input-output relations in a NN [3, 91]. To understand the impact of MLP structure, for example number of layers, on power saving for MLP inference, MNIST is trained with 3-layer MLP (784-144-10) and 4-layer (784-

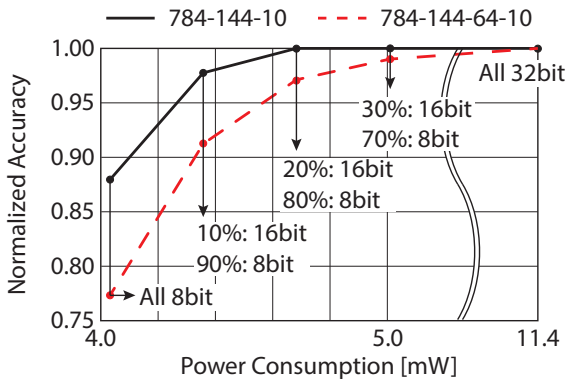


Figure 3.11: Comparing MNIST trained with 3 layers (784-144-10) and trained with 4 layers (784-144-64-10) in terms of power and accuracy.

144-64-10). For both cases, maximum number of iterations is 3,000 and 32bit is used for training. Inference is also performed with same MLP structure used in training. As both cases show similar accuracy for inference with 32bit ($\sim$0.977), I can say that 4-layer MLP is not the optimal MLP structure for MNIST, in other words, it has a redundant layer.

Fig. 3.11 shows 3-layer MLP can save more power than 4-layer MLP for all three approximations. For example, I assumed that target accuracy is 0.95. Training with 3 layers can use 16 bits for 10% synapses and 8 bits for 90% synapses while 16-bit precision is used for 20% synapses and 8-bit precision for 80% synapses with 4 layers. According to [92], MLP structure with more layers is more sensitive to weight error if the number of synapses for single neuron is high enough. Unlike Convolutional Neural Network [9], which has local neighbor connections, MLP with many layers could be more sensitive to weight error; less power saving during the inference. Training with less number of layers (optimal network structure) is critical to not only training time (less number of computations) but also power saving during the inference.

For approximation in deep network (network composed of many layer), more advanced approximate techniques are required to compensate accumulated error through layers. First, to avoid shrinking value due to approximation, normalization at each layer is required [93]. Moreover, stochastic rounding [93] (round up or round down randomly) can have both pos-

itive and negative quantization error in network. In this chapter, for simplicity in hardware design, such advanced approximating units are not considered. In addition, as I already explained in 3.1.1, layer-wise approximation (approximation more neurons in earlier layer) could be considered to improve the accuracy while maintaining same power saving.

3.2.4 Summary of Relationship Between Training Conditions and Power Saving

Fig. 3.12 illustrates the average power consumption for single PE and its normalized accuracy for different approximate approaches (proposed algorithm, software approach, and hardware approach) and different training conditions. For hardware approach, the ratio of in-exact multipliers increases from 10% to 100%. For all different training conditions, proposed algorithm shows the best power saving, and hardware only (in-exact multiplication) approach shows the least power saving for target accuracy.

3.2.5 Retraining with Approximate Synapses

In this section, I will discuss about retraining the network after approximation to improve the accuracy while maintaining the power consumption during on-chip inference. Fig. 3.13 shows the flow of retraining. After initial training, error sensitivity is computed by gradient descent. Under the target accuracy (or accuracy degradation margin), bit-precision rate (e.g., [rate for 32bit, 16bit, 8bit] = [0%, 20%, 80%]) is determined by proposed algorithm. After approximate synapses using the bit-precision rate, neural network is retrained with a small training data set with few iterations. During the retraining, approximated synapses maintain its bit precision. In other words, I allowed quantization error during the retraining. Although retraining requires additional latency and energy consumption, it will be performed once before on-chip inference. The inference will be repeated many times. For 50 epochs of training of MNIST, energy consumption is $1.69mJ$, which is similar to inference 263 times ($263 \times 6.41\mu J$). Therefore, I believe power saving during the inference while yielding high accuracy is desirable, even at the expense of increased training energy.

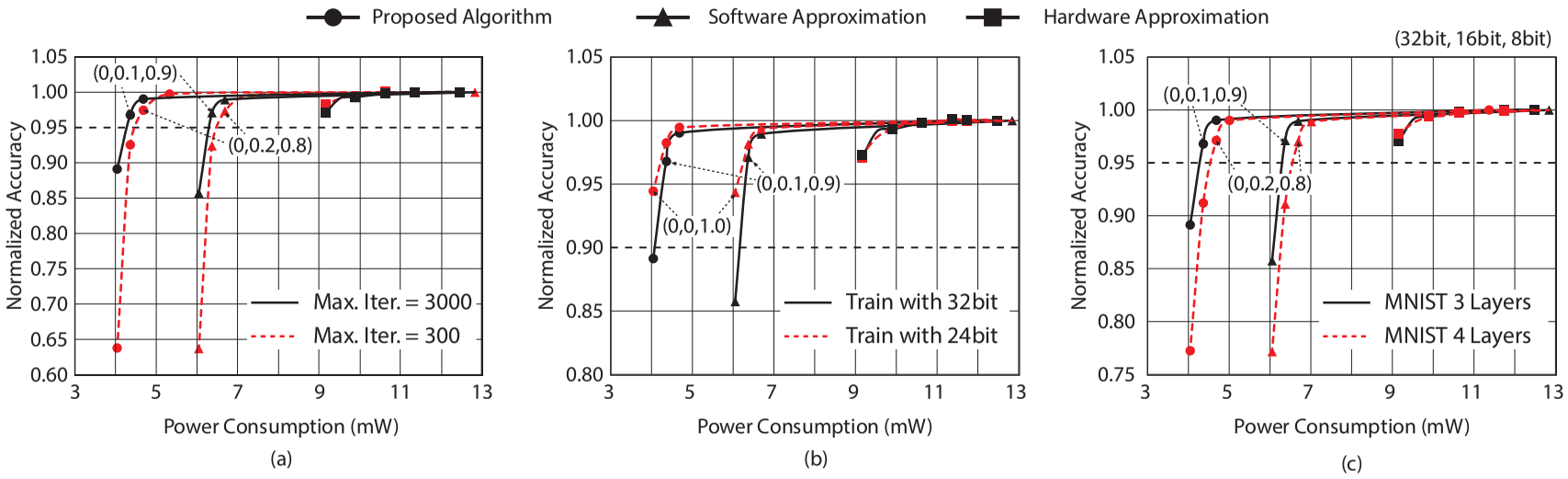


Figure 3.12: Analysis of the impact of different training conditions on the power saving during the testing based on the proposed algorithm. (a) different maximum number of iterations, (b) different bit precision, and (c) different MLP network structure. Software approximation changes the bit-precision while using only accumulate multiplier; Hardware approximation changes the ratio of inexact PE from 10% to 100% while using 32-bit full precision; Proposed algorithm uses 40% inexact PE and changes the bit-precision.

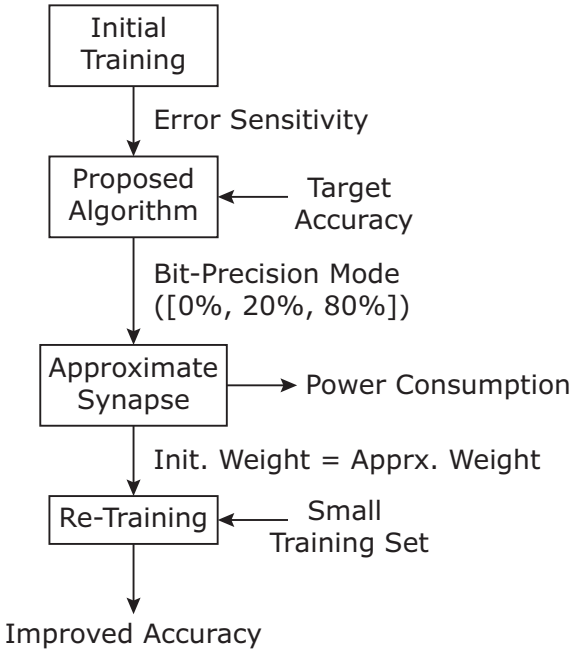


Figure 3.13: Retraining flow. After approximate synapse using proposed algorithm, bit-precision for each synapse is maintained. Therefore, accuracy is improved while power consumption is constant.

Fig. 3.14 shows the accuracy improvement by retraining. I constrained the maximum iteration for retraining as 50; it is very small number of iterations compared to number of iterations in initial training ($\sim$3,000). Base accuracy (before retraining) is determined as 20% 16bit and 80% 8bit approximation. I should mention that the power consumption of the MLP hardware is constant since bit precision for synapse is maintained. When initial training is iterated 3,000 times, base accuracy is 0.9676 and its power consumption is 4.686W. After 15 iterations for retraining, accuracy is improved up to 0.9731. The impact of retraining is much significant when initial training is less iterated. After initial training with 1,000 iterations, base accuracy is 0.9497, but it can be improved to 0.9643.

Note that the inference accuracy can also be improved by allowing less approximation during inference, instead of using retraining. However, that will increase power dissipation. For example, consider the case of initial training with 3,000 iterations. The improved accuracy of 0.9731 with retraining can also be achieved with 5.007W (+7%) power consumption during inference without retraining. Similarly, in the case of training with 1,000

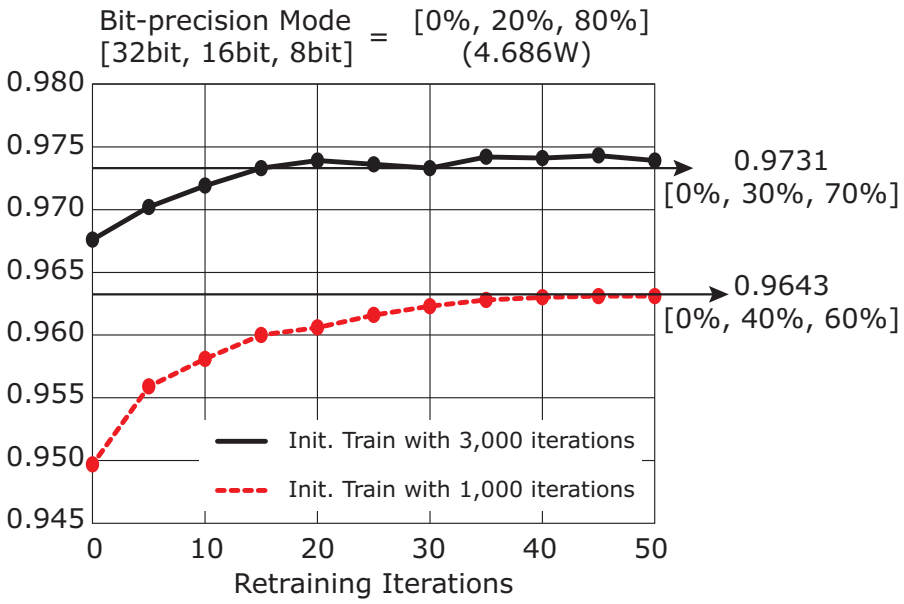


Figure 3.14: Accuracy improvement by retraining while maintaining power consumption for different initial training conditions.

iterations, I need about 5.328W (+13%) power for inference without retraining to achieve the accuracy level of 0.9643. In summary, re-training after approximate synapses with small number of iterations (50) is effective method to improve the accuracy without any power penalty. Especially, if Algorithm 1 cannot guarantee both quality constraint and power constraint, retraining should be considered.

3.2.6 Comparison Proposed Algorithm with Uniform Bit Selection

In [79], network is trained without any approximation initially. After initial training, network is retrained with weight restriction (approximation). Since all synaptic weights are encoded in same bit-precision, in this section, I refer it as 'Uniform Bit Selection'. According to [79], accuracy of MNIST with 8-bit MLP is about 0.9745. To compare the proposed algorithm with uniform bit selection, uniform bit selection approach is applied in digital MLP full system (100% accurate MAC with 8-bit precision).

Fig. 3.15 compares two approaches in terms of normalized power and MNIST accuracy. From the proposed algorithm, I can get 0.9644 for MNIST accuracy with 64% power sav-

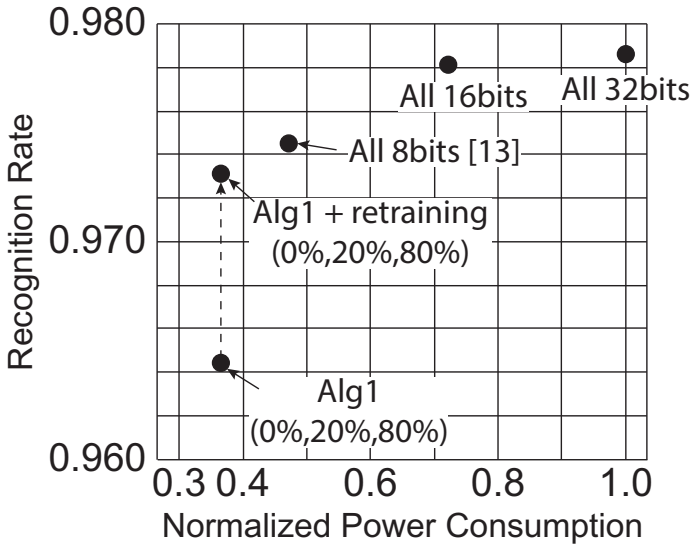


Figure 3.15: Comparison proposed algorithm (with retraining) with uniform bit selection.

ing. After retraining explained in Section 3.2.5, I can get 0.9731 without additional power consumption. Although the accuracy of proposed algorithm is slightly lower (-0.0087) than uniform bit selection, it can save more power (-11%).

Since MAC is replaced with alphabet set multiplier (ASM) with limited number of alphabets [79] to save dynamic power consumption, coupling bit precision control based on error sensitivity with ASM remains as future work.

3.3 Fixed Point with High Accuracy

Recent previous work shows that 16 bit fixed point precision is enough for inference of deep learning [27, 28, 94, 95]. However, for training, even 32bit fixed point is not enough to train deeper network, such as RNN [96]. To overcome *quantization* error in fixed point, stochastic rounding is applied [93, 96].

Fig. 3.16 shows cost function of network during the training RNN. 32bit fixed point with stochastic rounding shows almost similar training trends with the trends of floating point case, while 32bit fixed point shows huge fluctuation in cost function and cannot converge as the result of floating point. It shows that using 32bit fixed point is not enough to

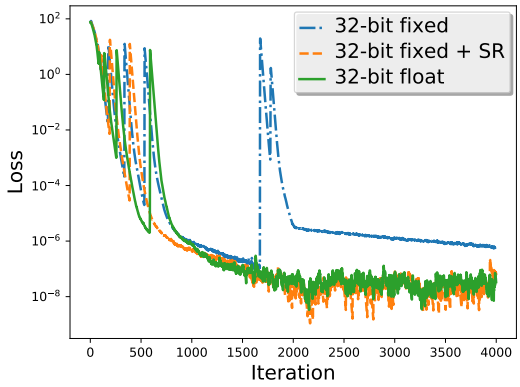


Figure 3.16: Training accuracy for RNN with different numeric representation.

train the deep network.

In terms of hardware, four different MAC units are designed: 32bit floating point, fixed 32/16bit precision, Fixed 32/16 with stochastic rounding (SR), and Fixed 32/16 with stochastic rounding (SR) low overhead version.

1. Fixed 32/16 MAC is designed to operate in two different precision modes: compute a pair of 32bit fixed point operands or two pairs of 16 bit fixed point operands in a single clock. Both 32bit and 16bit fixed point has 4 bit integer part and 28/12 bit for fractional part. Fig. 3.17 shows operation multi-precision MAC operations. During the training, 16-bit state and weight are zero-padded and applied to the MAC. From 64-bit output, 32 bit output is cropped while maintain 4bit integer part. The result has 24bit fractional part (last 4 bits are zero). During the inference, two pairs of 16bit operands are delivered to MAC arrays. The result is two 32-bit outputs. It is cropped to two 16-bit outputs while 4-bit integer is maintained.

2. Fixed 32/16 + SR To add stochastic rounding, 64 random number generators are added [96] to generate 64 bit random number per each clock. However, the overhead of 64 random number generators is too high so its area and power is almost similar to that of floating point unit.

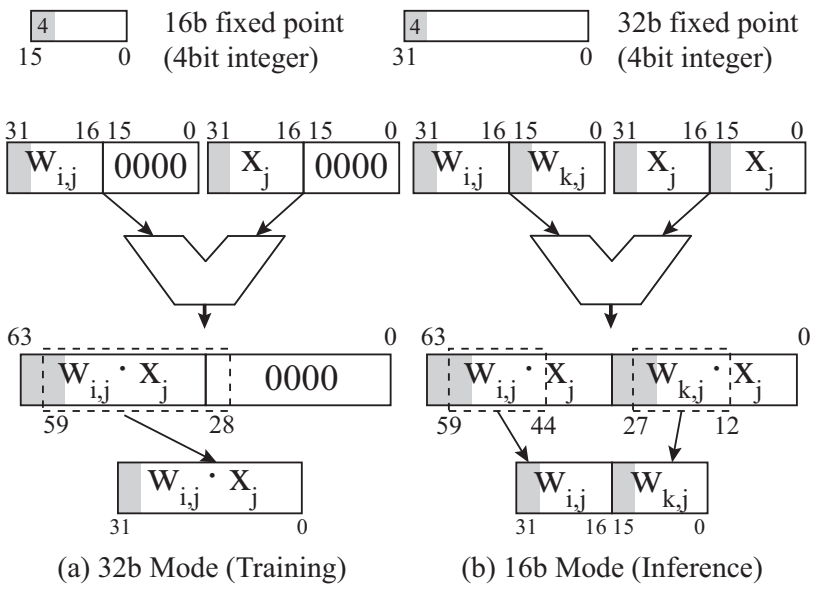


Figure 3.17: Multi-precision MAC operating mode (a): 32bit mode and (b): 16bit mode.

3. Fixed 32/16 + SR LO To reduce power/area overhead, a single random number generator is used and it generates a single bit in every clock (Fixed 32/16 + SR LO). It feeds to LSB of 32bit left shift register RAND register (Fig. 3.18). After initial 32 cycles, RAND will deliver 32bit lenght random value in every cycle. By reducing the overhead of stochastic rounding unit, its power and area overhead is smaller than using floating point unit about 30%. .

3.4 Conclusion

This chapter discussed the concept of approximate synapse to reduce power dissipation of a feedforward network, namely, MLP, during inference. The approximated synapses are selected based on gradient (error sensitivity of synaptic weights) precomputed during training phase. I observed that training conditions play an important role in power saving during the inference with approximate computing. For example, training with reduced bit precision or more iterations provides more opportunities for power saving with approximate synapses during inference. However, use of optimal number of layers in the MLP is critical; too many layers increase the sensitivity to small error thereby reducing the effectiveness of approx-

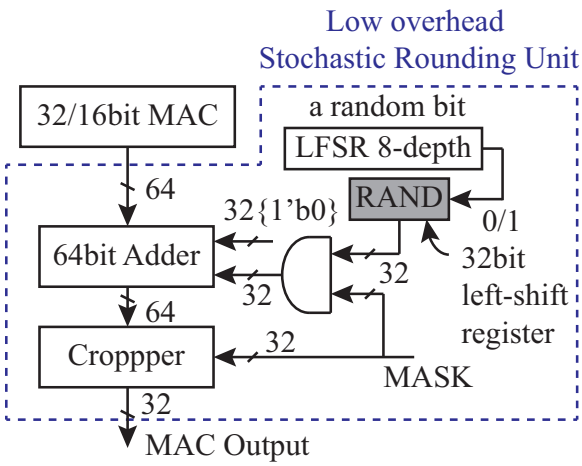


Figure 3.18: Fixed 32/16bit MAC with low overhead stochastic rounding unit: a single LFSR and 32bit left shift register.

imation. Further, it is shown that after the selection of approximate synapses, performing retraining can improve the accuracy; hence, more aggressive approximation be-comes possible for a target accuracy. In conclusions, the approximate synapse is observed to be an effective technique for reducing power dissipation during inference; however, the effectiveness depend on the training conditions. Moreover, for on-chip training, fixed point MAC unit with stochastic rounding using *one* pseudo random number generator is introduced. 32 bit fixed point with stochastic rounding can save about 25% of area, power overhead compared to single precision floating point, but shows similar training accuracy.

CHAPTER 4

NEUROCUBE: DIGITAL DEEP LEARNING ACCELERATOR BASED ON 3D STACKED MEMORY

In this chapter, basic architecture of NeuroCube is introduced targeting only *inference* of deep learning.

Fig. 4.1 shows the structure of hybrid memory cube (HMC). To utilize all 16 vaults effectively, each vault needs to be connected to one PE placed on the logic die. Note that in general I can have a different number of MAC units/PE to match the vault bandwidth. All PEs are connected by a 2D mesh network.

Fig. 4.2 illustrates the key components of NeuroCube architecture designed in the logic die of a HMC. Multiple processing elements (PE) concurrently communicate with multiple DRAM vaults through high-speed TSVs. A host communicates with the NeuroCube through external links of the HMC to configure (program) the NeuroCube for different neural network architectures (such as number of layers, types of layers, and dimension of layers). NeuroCube architecture is composed of a global controller, programmable neurosequence generator (PNG) for DRAM, routers for a 2D-mesh network on chip, and processing elements (PEs). Notations to describe the architecture are explained as: Number of DRAM banks (vaults or channels): n_{Ch}, Number of PE per DRAM bank (vault): n_{PE}, Number of MACs per PE: n_{MAC}, DRAM I/O clock frequency: $f_{DRAM-IO}$, NoC router clock frequency: f_{NoC}, PE clock frequency: f_{PE}, MAC clock frequency: f_{MAC}.

4.1 Architecture Design

The processing element (PE) is the main computing unit and is comprised of multiple multiply accumulator (MAC) units since weighted summation is the main arithmetic operation in the emulated NNs (Eq. 2.1). A single PE is composed of n_{MAC} MAC units, a cache

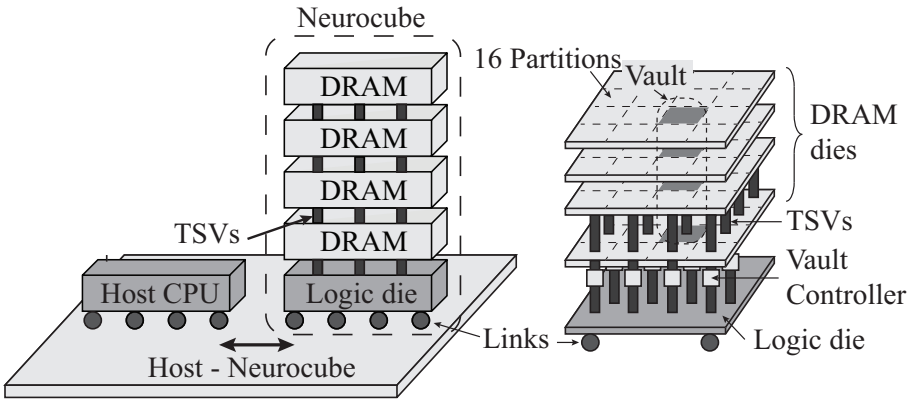


Figure 4.1: NeuroCube architecture based on Micron's Hybrid Memory Cube [58, 60] communicating with host.

memory, a temporal buffer, and a memory module for storing shared synaptic weights. Fig. 4.2 (b) shows a block diagram of a single PE.

4.1.1 Multiply-Accumulator

In this section, 16-bit fixed point is (($Q_{1,7,8}$: 1bit MSB, 7bits integer, 8bits fractional part)) used for both the state and weights in a generic neural network. As training is not considered in this chapter, simple fixed 16 bit fixed MAC units are placed in NeuroCube (No stochastic rounding or no 32bit operation)The operating clock frequency of a MAC (f_{MAC}) is determined as below

$$f_{MAC} = f_{PE}/n_{MAC} \tag{4.1}$$

where f_{PE} = f_{NoC} = $f_{DRAM-IO}$ (operating frequencies of the PE, NoC, and DRAM I/O system). To compute sum of multiplications ($\sum W \times X$), the output of a MAC needs to be used as an input in next cycle. (Fig. 4.2 (b)).

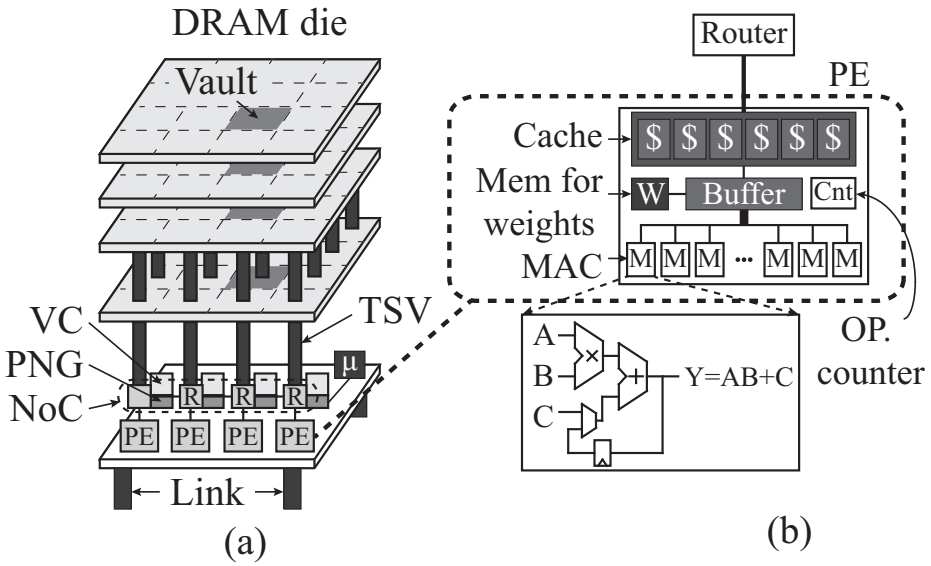


Figure 4.2: (a) NeuroCube architecture and (b) Organization of the processing elements (PEs).

4.1.2 PE Memory

The operation of the PE and the role of the various memories are best illustrated with an example. Consider a network where each layer has 8 neurons and each neuron has 3 input neurons from the previous layer. The PE has 8 MAC units. Consider the update of a layer. The 8 MAC units in a PE synchronously process/update one output neuron at a time. On cycle 1, each MAC unit computes the summation with the first input of each neuron. On cycle 2, each MAC unit computes the second term of the summation using the state and weight from its second input neuron and so on. Thus in three cycles the states of all 8 neurons have been updated.

The state of the input neurons and their associated connectivity weights are encapsulated in a packet and moved to the PEs by the programmable PNGs. Each packet has an OP-ID value to indicate whether they are the first, second, etc. input for its corresponding output neuron. Each PE has an operation counter (OP-counter) used to sequence through the correct number of input neurons for each output neuron whose state is being updated. At any point in time, the OP-counter refers to the input number currently being computed

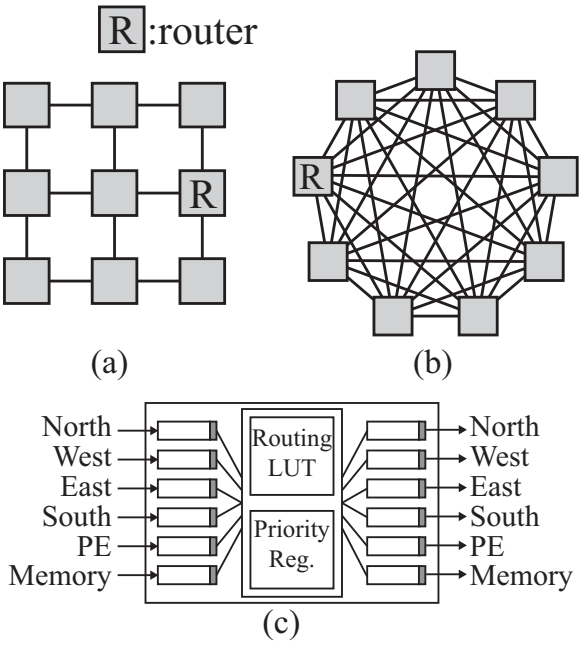


Figure 4.3: (a) 2D mesh NoC, (b) 2D fully connected NoC, and (c) Router design for 2D mesh NoC.

by the MAC units (i.e., input 1, input 2 etc.). If packets arrive out of order they are buffered in the SRAM cache until all corresponding inputs arrive, i.e., it is buffered if the OP-ID of the packet is greater than the OP-counter value. When all corresponding inputs arrive, they are moved to the temporal buffer and the availability of all inputs triggers a MAC operation. Finally, if the size of synaptic weights matrix is small all weights are stored in PE weight memory.

4.1.3 2D Network on Chip

The PEs are interconnected by a 2D mesh network as shown in Fig. 4.3 (a). Fig 4.3 (c) illustrates a block diagram of the router. Each PE is connected to a single router. Each router has 6 input channels and 6 output channels (4 for neighbhoring routers and 2 for PE and memory). The router is wormhole switched with credit-based flow control, a 16-depth packet buffer for each input and output channel, and table-based routing. Routing is deterministic X-Y routing. Input buffers use a rotating daisy chain priority scheme for arbitrating between inputs requesting the same outputs. Priorities are updated every clock

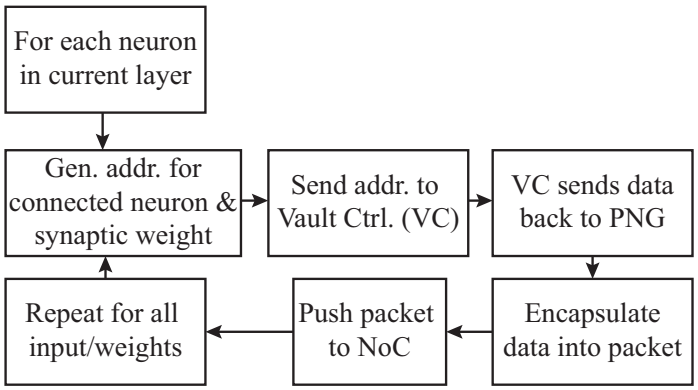


Figure 4.4: Operation of the Programmable Neurpsequence Generator (PNG) for computing one layer.

cycle.

4.2 Memory Centric Neural Computing

In this section, I describe the design/programming of the Programmable Neuroseuquence Generator (PNG). The host programs the execution of one layer at a time. For example, NeuroCube to operate ConvNN for scene labeling [38] with six layers (2Dconv, pooling, 2Dconv, pooling, 2Dconv, and fully connected) where different layers demonstrate different types of connectivity (local connection in 2Dconv, and all to all connectivity, similar to MLP in fully connected layer) should be programmed six times.

The execution of each layer is described in this section as i) the packetization and flow of data between memory and the PEs, ii) the addressing of memory and programming of the memory-based state machines, and iii) the programming interface to the host.

4.2.1 Orchestration of the Data Flows

Each vault controller in the HMC has an associated programmable neurosequence generator (PNG) that controls the data movements required for neural computation. Fig. 4.4 shows the operation of a PNG for each layer in the NN. For each neuron in the layer, the PNG will generate the addresses of the connected neurons in previous layer and weights in

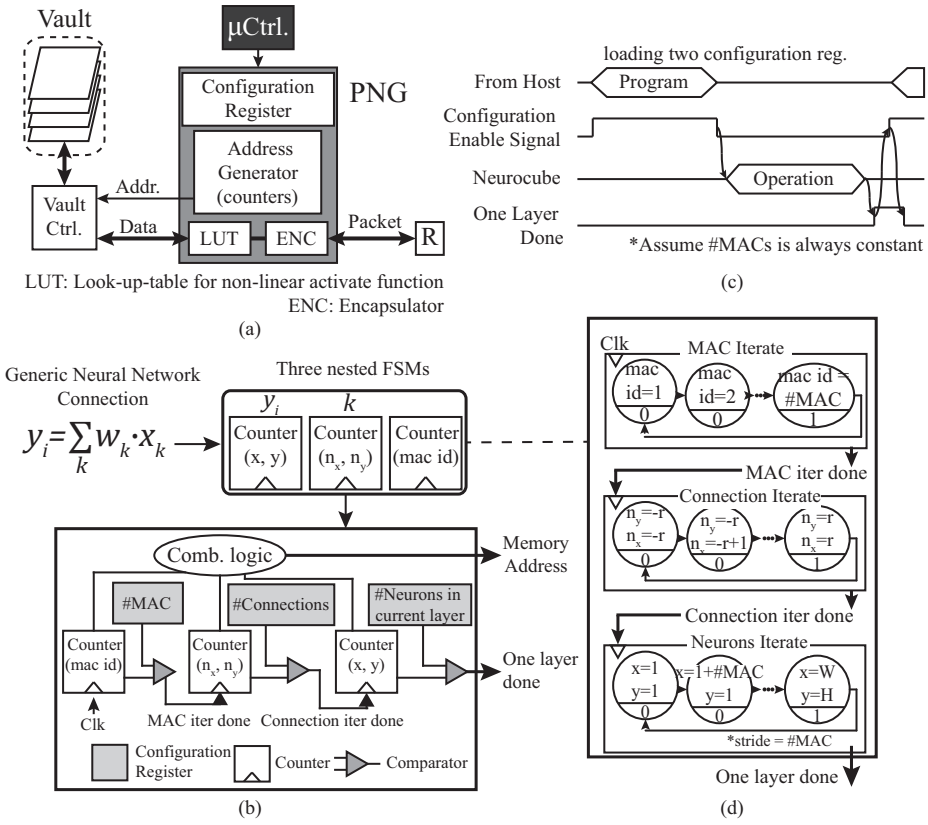


Figure 4.5: (a) Interaction between PNG, VC, and host. (b) General neural network can be translated as three nested loops, which can be implemented using three counters. (c) Timing diagram of programming PNG and NeuroCube operation. (d) Three nested loops can be mapped to finite state machines.

the memory. Consider a neuron in one layer (y_i in Fig. 4.5 (b)). To compute the state of this neuron the PNG generates a sequence of addresses for the locations of (a) the state of all connected neuron (x_k) and (b) the corresponding synaptic weights (w_k) between them. The preceding is repeated for each neuron in the network. The PNG executes this operation and sends the address sequences to the vault controller (VC).

As the PNG receives the data stream from the VC, the data corresponding to each connected neuron is encapsulated into a packet. The PNG encodes a specific MAC-ID for each packet such that all packets corresponding to the neuron and its connected neuron have the same MAC-ID. The PNG also pushes states (y_i in Eq. 2.1) through the non-linear activate function (implemented as the Look Up Table (LUT)) and the output of neuron (x_i in Eq. 2.2) is embedded in the packet. Finally, the packet includes source ID (memory vault

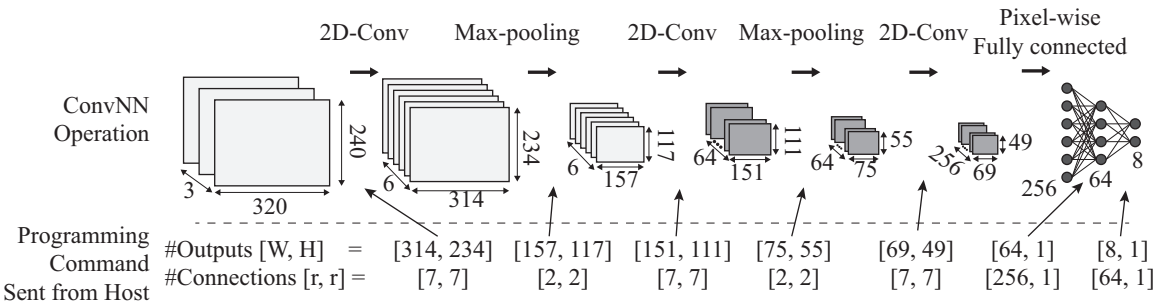


Figure 4.6: Convolutional neural network for scene labeling [38] and programming parameters for each layer

ID) and destination ID (PE ID) and is injected into the router in the NoC to deliver to the PEs. (Fig. 4.5 (a)).

After the MACs finish the computation, the state of output neuron is encapsulated into packets for each MAC (all MACs finish the operation at the same time) with MAC$-$ID and injected into the network. When the PNG receives the packet (write back), the PE index (SRC in the packet and MAC-ID are sufficient to determine the neuron to be updated and its address. This information is pushed back to the VC.

4.2.2 Design and Operation of the PNG

Fig. 4.5 (a) shows the block diagram of the PNG, composed of i) the address generator, ii) configuration registers, iii) a Look-Up-Table (LUT) for the non-linear activate function, and iv) packet encapsulation/de-encapsulation logic. Each PNG is programmed by a global controller (μ ctrl.) interacting with the host. The host loads configuration registers (see below) to initiate the computation of a single layer. After all 16 PNGs are configured, computation begins.

The *address generator* in the PNG is designed as a programmable finite state machine (FSM) that can be used to sequence through addresses for single layer of neurons. Operationally, the computation over a single layer of neurons is composed of three nested loops: a loop across all neurons in the layer, a loop across all connections for single neuron in

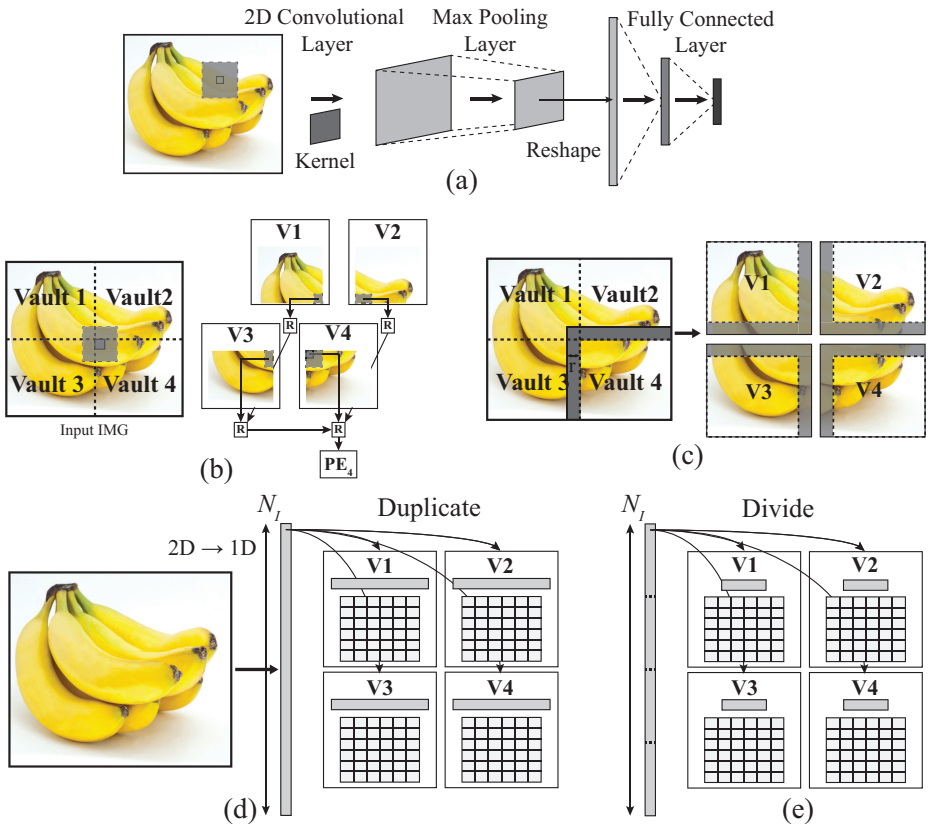


Figure 4.7: Data movement in NeuroCube (assume 4 vaults and 4 PEs). (a) ConvNN structure. (b) For 2D convolutional layer, input image is divided into 4 non-overlap segments. (c) To reduce NoC traffic, input is divided with overlapped area. (d) For fully connected layer, input image is transformed to vector and this vector is duplicated to all HMC vaults. (e) To reduce duplicated memory overhead, input vector is divided into all vaults.

the layer, and a loop across all MACs. A single MAC computes the state of one neuron at a time; therefore n_{MAC} MACs compute n_{MAC} neurons after iterating over $n_{Connections}$ computations (multiplication and additions). This process is repeated until the state of all neurons in this layer have been computed. The FSM structure using three counters (one for each loop) is shown in Fig. 4.5 (b). These counters are programmed by the host to initiate the computation of each layer.

The *combinational logic* computes the memory address of each required connected neuron and synaptic weights for current neuron in this layer. This logic receives the current states of the *neuron counter* (cur_x, cur_y) and *connectivity* (n_x, n_y), and computes the target

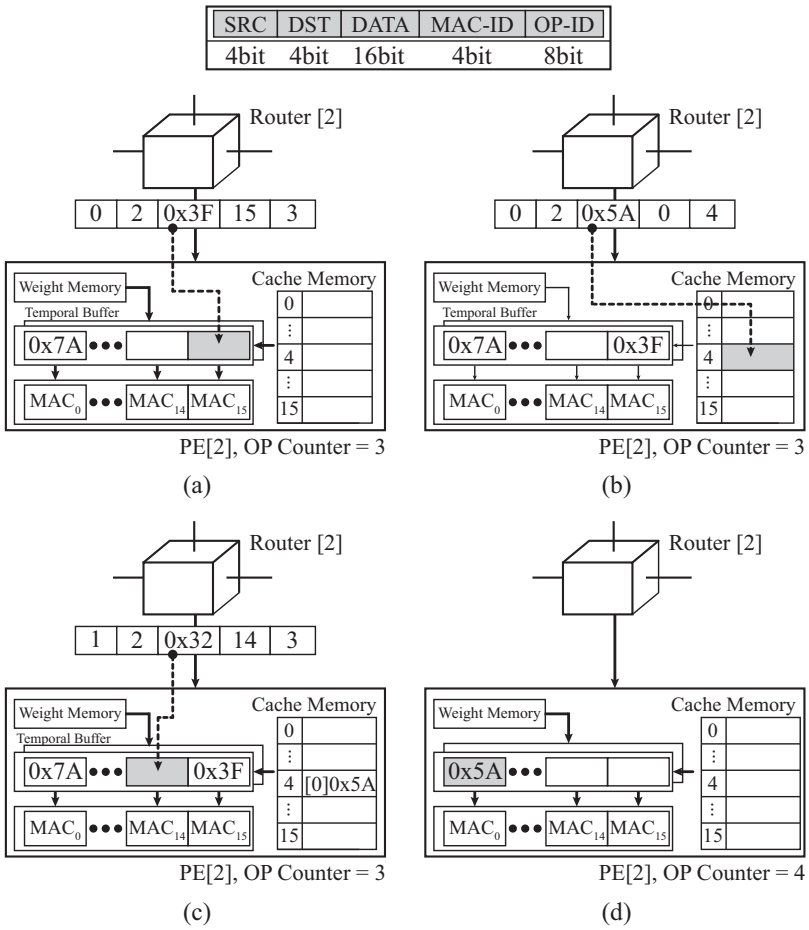


Figure 4.8: Operation of PE (OP counter = 3). (a) Packet with OP-ID as 3 arrives PE and moves to temporal buffer. (b) Packet with OP-ID as 4 arrives PE and moves to cache memory. (c) Packet with OP-ID as 3 arrives PE and moves to temporal buffer. Temporal buffer receives 16 weights and input (d) Buffer is flushed and MACs start computation. Operation counter increases. Before start 4^{th} operation, bring pre-stored data from cache memory.

address $(targ_x, targ_y)$ as follows:

$$targ_x = cur_x + n_x, \quad targ_y = cur_y + n_y, \tag{4.2}$$

Note (cur_x, cur_y) is the coordinate of the current neuron. Based on the the number of connected neurons and their weights, the memory range allocated for this layer (state of connected neurons and synaptic weights) can be precomputed. Therefore, the actual mem-

ory address for the required neuron located at $(targ_x, targ_y)$ is computed as:

$$Addr = targ_y \times W + targ_x + Addr_{last}, \tag{4.3}$$

where W represent width of output image and $Addr_{last}$ represents the last memory address from the previous layer.

Fig. 4.5 (c) shows timing of host-PNG interactions. To program the PNG, the host asserts a configuration enable signal to write configuration registers (see next section). After the host writes all configuration registers, it deactivates the configuration enable signal that initiates the FSM. Fig. 4.5 (d) shows that how the three counters inter-operate. When the state value of the *current neuron* counter equals the total number of neurons in this layer, it means that the PNG has generated all data address sequences required for this layer. After generating all required data addresses, the PNG waits to receive the newly computed state for the last output pixel. After it receives the last state, the PNG raises the layer done signal. Then host now starts programming the next layer.

4.2.3 Programming of the PNG

Programming the PNG initially map all data structures of NN (e.g., input image and weights) into the physical address space of the cube (i.e., across vaults, dies, banks etc.) followed by periodic configuration to process each layer of the neural net. The implementation of this global controller can be via software executing on a simple micro controller on the logic die or, directly by the host via the HMC links. In the latter case the configuration registers must be accessible to the host. In this chapter, I assume the latter, i.e., direct host programming.

As an example, consider how the PNG can be configured for a convolutional NN during inference (Fig. 4.6). The number of MACs is determined by design as 16. To iterate over all neurons in the 1^{st} convolutional layer, the configuration register for the number of neurons should be set to 73,476 (314 $\times$ 234). The value of the counter for iterating over current

neurons is incremented by 16 at each step since the states of 16 neurons in this layer are computed simultaneously. For one neuron, the number of connection is 49 (7×7), and is also programmed into the PNG.

4.3 Compute Operation in NeuroCube

4.3.1 Management of Data Movement

Application: ConvNN for Scene Labeling

In this section, I discuss how to reduce data movement, specially over the 2D NoC, for locally connected and fully connected layer. Fig. 4.7 shows that how input image and weight matrix are divided into n_{Ch} partitions and stored in DRAM vaults to manage data movement.

1. Locally Connected Computation: For small connections, the weights are duplicated in the weight memory of all PEs. Only input needs to be partitioned and stored in the vaults as illustrated in Fig. 4.7 (b). The 2D NoC traffic can be reduced by dividing into multiple overlapped image segments (Fig. 4.7 (c)). The overlapping can improve throughput with small memory overhead, specially for small kernels; the memory overhead increases with kernel size.

2. Fully Connected Layer: I can divide the weight matrix into DRAM banks (Fig. 4.7 (d)) and duplicate the input vector in each vault so that a PE can compute neurons using data from a single vault. However, when input image is too large to duplicate, both input and weight matrix need to be divided resulting in higher NoC traffic (Fig. 4.7 (e)). Note a fully-connected model can be used to represent *irregular* connections between neurons by storing a synapse weight of '0' for missing connections.

4.3.2 Operation of the PEs

Fig. 4.8 illustrates PE operations after programming by global controller. Fig. 4.8 (a) shows data and packet transfer among vault, PNG, and router. From a single DRAM vault

in HMC-Int., the PNG receives 32bit data and encapsulates that into two packets. Source (SRC) indicates the DRAM vault (4bit for 16 vaults in HMC-Int.), and destination (DST) indicates PE (4bit for 16 PEs). As all MACs operate in parallel, n_{MAC} weights and n_{MAC} states are delivered to PE. Each packet has a MAC-ID (4bit for n_{MAC}=16) to represent the target MAC. Each packet has OP-ID to represents the sequence of operations to compute one single output neuron. I assign 8bit for OP-ID. If maximum iteration for one pixel is more than 256, OP-ID represents the remainder of OP-ID divided by 256.

Fig. 4.8 (a)-(d) illustrates example when PE needs to compute 3^{rd} operation (OP-counter is 3). Packet for MAC_{15} to operate 3^{rd} operation is moved to temporal buffer[15] directly (a). If packet's OP-ID is higher than current OP-counter, it moves to cache memory (b). Cache memory is divided into multiple sub-banks (e.g. 16 sub-banks as in Fig. 4.8). When a new packet arrives which is not for the current operation (OP-ID != OP-counter), it is stored in sub-bank mod(OP-ID, 16). When temporal buffer receives all 16 input pixels and 16 synaptic weights, temporal buffer is flushed, MACs start computation, and PE increases OP-counter (c). For next operation, pre-stored data in cache memory is moved to temporal buffer (d). When checking for necessary packets for the next operation, the PE performs a full search of the sub-bank that may take anywhere from 16 clock cycles (16 MACs) to 64 clock cycles (max 64 entries).

4.4 System Throughput Simulation

I developed a cycle-level NeuroCube simulator for performance evaluation. In simulator, main memory specification (bandwidth, number of channels, bus-width), PE (cache size, number of MACs), and router (buffer size, latency) are parametrized. For all 16 vaults in the HMC, 32bit word (2 data items) is pushed at 5GHz (HMC specification) in burst mode and burst length is assumed as 8. Therefore, after pushing 8 words, the HMC needs to wait t_{CCD} before sending the next 8 words. Reference clock in the simulator is the main memory clock frequency (2.5GHz $\times$ 2 = 5GHz).

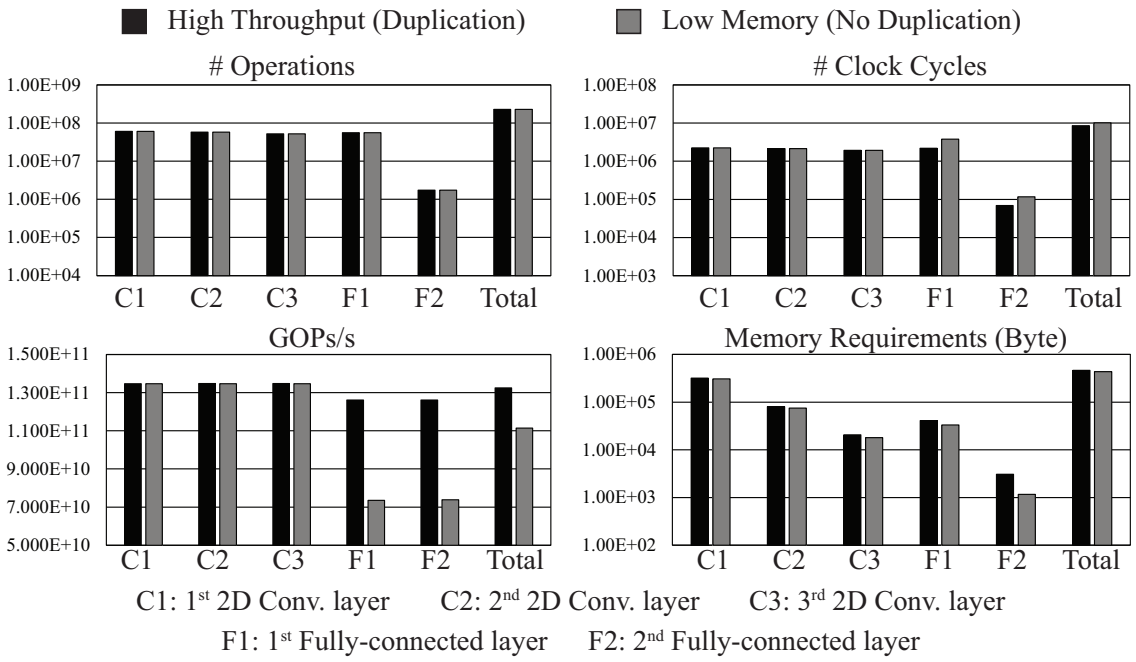


Figure 4.9: NeuroCube performance for scene labeling [38]. NeuroCube can operate high throughput with data duplication (black) or slightly lower throughput to save memory requirement (gray). (a) Number of operations, (b) Number of clock cycles, (c) Throughput (GOPs/s), and (d) memory requirements and overhead for inference with data duplication

The system simulation is performed considering a multi-layer ConvNN structure for scene labeling [38]. I choose ConvNN as the example, because it helps demonstrate the programmability of the NeuroCube for locally connected (2D convolutional) as well as fully connected networks. This ConvNN is constructed with 7 layers and input image is RGB 320 $\times$ 240. The number of neurons for each layer is illustrated in Fig. 4.6. I analyze the system performance during both inference and training, while most of the prior hardware works only considered inference ([97][21] [22] [25][23]) or training for simple application like MNIST (2 layers, 28$\times$28 input, [24]).

Fig. 4.9 shows the throughput and memory requirements for inference operations in scene labeling [38].

The three convolutional layers and the first fully connected layer dominates the number of operations. In particular, the former dominates execution time in training. With data

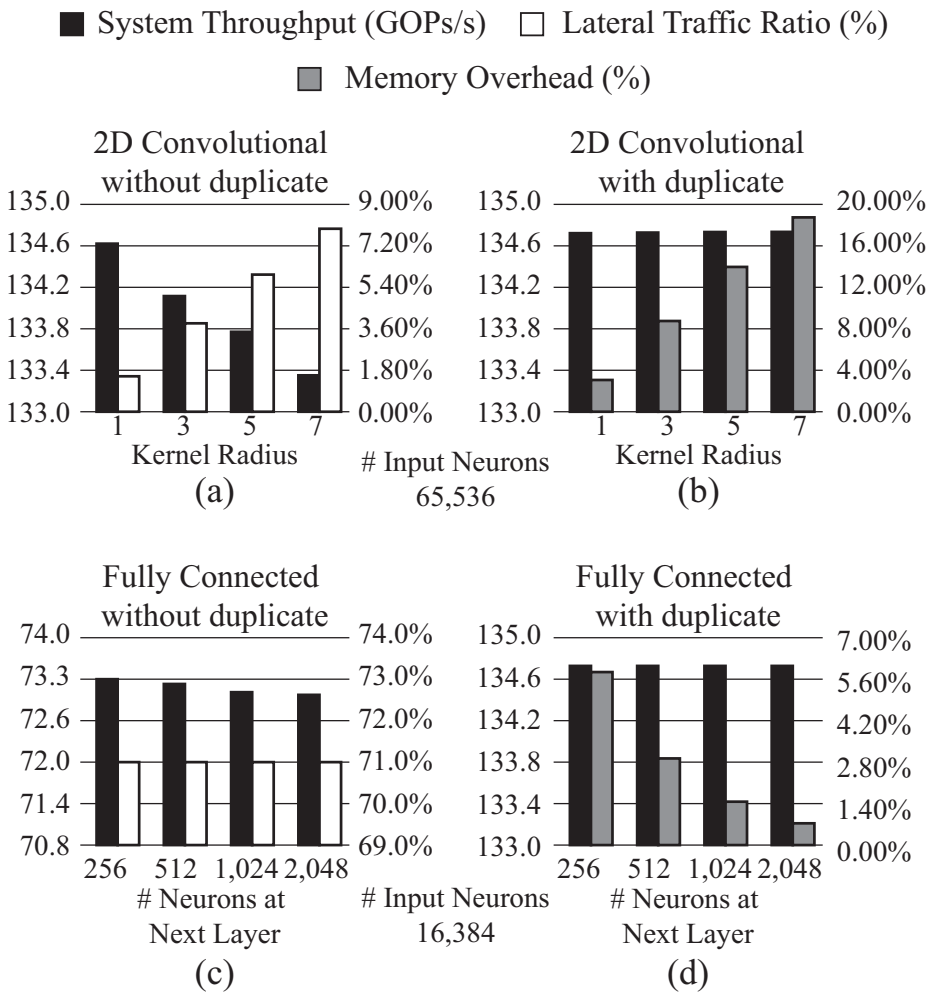


Figure 4.10: Effect of NN parameters on throughput and memory: effect of kernel size in 2D convolutional layer (a) without duplicate and (b) with duplicate; and effect of number of neurons at in the hidden layer for fully connected layer (c) without duplicate and (d) with duplicate.

duplication (black bar), NeuroCube shows almost constant throughput since there is no lateral traffic for both types of layers (132.4GOPs/s). Without data duplication (gray bar), throughput for fully connected layers degrades; therefore total throughput is slightly lower than that of data duplication (111.4GOPs/s).

I estimate the image processing throughput for the scene labeling application based on the RTL level design of the NeuroCube hardware in 28nm and 15nm nodes. The estimated throughput for inference are 17.52 frames/second in 28nm and 292.14 frames/second in 15nm.

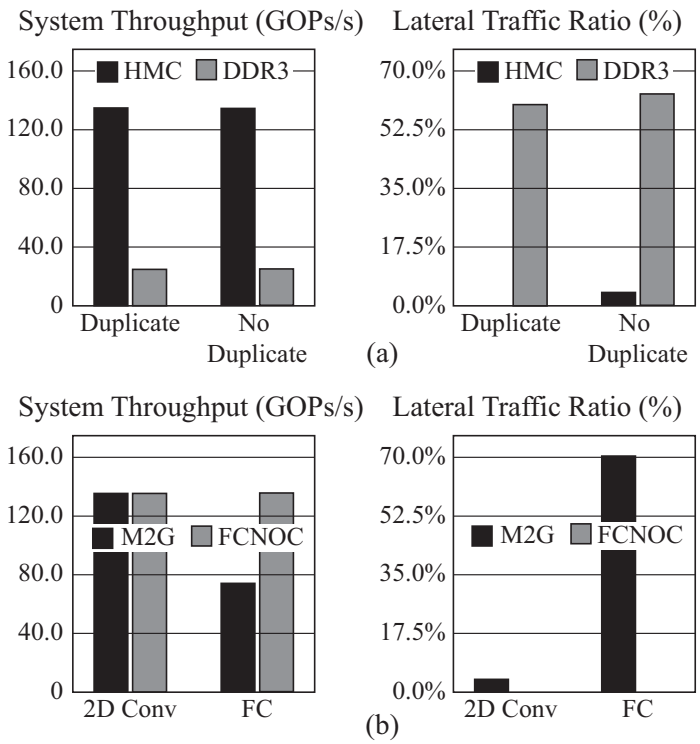


Figure 4.11: Performance comparison: (a) HMC and DDR3, and (b) mesh grid and fully connected NoC.

Extending NeuroCube for Other Neural Networks: The example of ConvNN shows how NeuroCube can be used to program other NN. Programming an MLP [36] or RNN [37] is similar to programming the fully-connected layer in ConvNN. Although RNN is not simulated in this chapter, RNN is equivalent to a deep MLP after unfolding in time, while LSTM [98], a variant of RNN with multiple hidden layers each with a different activation function, can be realized by updating the LUT for each layer during programming. On the other hand, programming a locally connected layer like Cellular Neural Network [99] is similar to programming the 2D convolutional layer. Therefore, different types of network can be programmed in NeuroCube without architectural changes.

4.4.1 Effect of NN Parameters

In this sub-section, I study the effect of NN parameters on the system performance. Fig. 4.10 (a) and (b) show the simulation results of the locally connected layer (2D convolutional

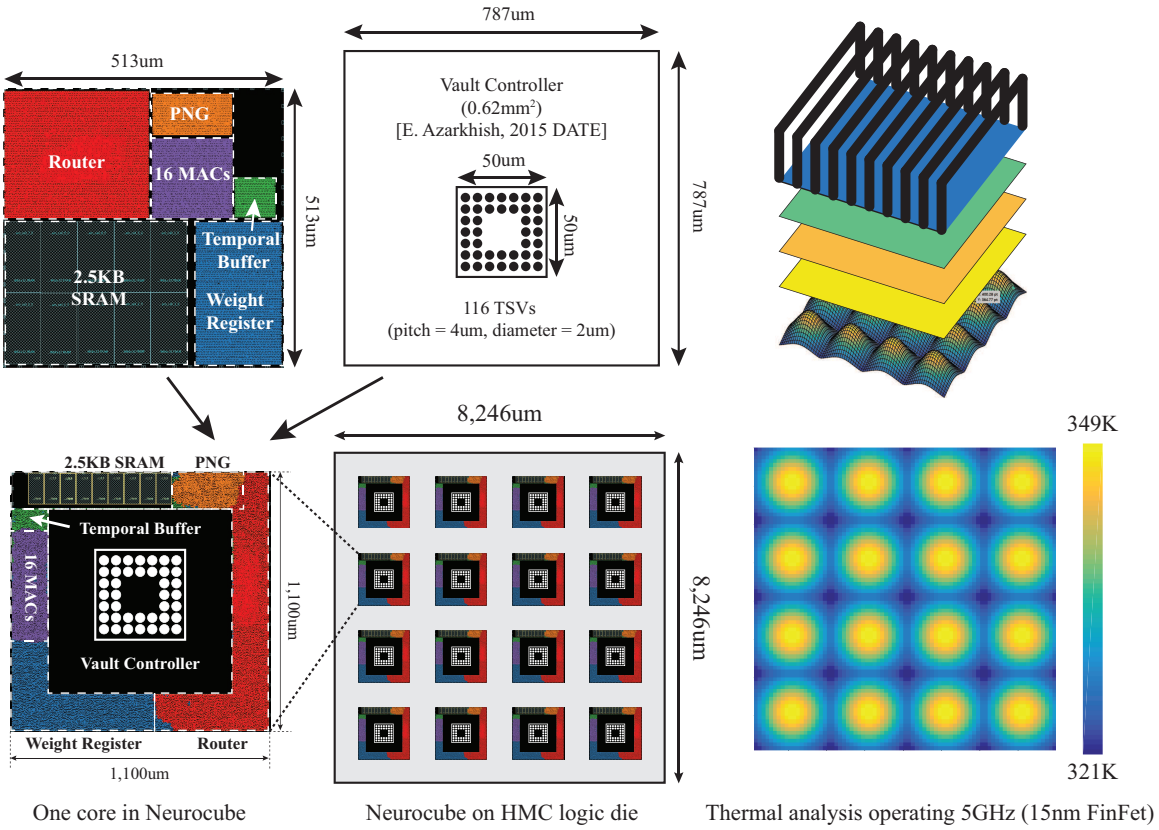


Figure 4.12: Layout of one partition of HMC logic die including a single PE (16 MACs, 2.5KB SRAM, weight registers, temporal buffer, and PNG), vault controller [62], and a single router. Maximum temperature of logic die is 349K and of DRAM die is 344K.

layer) with different kernel sizes. When each vault does not duplicate the neighborhood input pixels, a higher kernel size increases the lateral traffic on the NoC thereby decreasing system throughput. When the neighborhood is duplicated, there is no degradation in system performance with large kernel size, however, the memory overhead due to duplication increases.

Fully connected layers are placed as last layers of the ConvNN as classifiers. Consider a 3-layer fully-connected network with one hidden layer (between input and output). More neurons in the hidden layer allow more complex non-linear classification, but also require more computation and memory. Without duplication, one PE should access all vaults to compute one pixel at the current layer; therefore lateral traffic on the NoC is high (71%) (Fig. 4.10 (c)). However, the number of connected neurons in previous layer for one neuron is constant. In other words, the amount of lateral traffic is also constant. Therefore, system performance is almost constant (Fig. 4.10 (d)). When number of neurons at the current layer increases, synaptic weights ($N_I \times N_O$) occupy the most of the memory in terms of space since it is fully connected (it has a huge weight matrix). Therefore, the portion of duplicated input neurons in total required memory decreases (memory overhead decreases).

4.4.2 HMC-Internal vs. DDR3

According to Table 2.1, peak bandwidth of DDR3 (12.8GBps) is higher than that of HMC-Int (10GBps); therefore, a comparative analysis of the two will help understand the role of concurrency in the NeuroCube. For the 2D Conv. layer, simulation is performed to analyze the impact of the number of channels. Fig. 4.11 (a) shows that DDR3 shows much lower system performance, although DDR3 has higher peak bandwidth. Since DDR3 has only two channels, data traffic on the 2D NoC is a major bottleneck. Due to large lateral NoC traffic (about 60%), there is no benefit from duplication in 2D Conv. in terms of system performance. Under the same bandwidth, more slower channels can leverage NoC traffic; therefore it improves the system performance.

4.4.3 Mesh Grid NoC vs. Fully Connected NoC

In addition to memory with many channels, a fully connected NoC (all routers are connected each other, Fig. 4.3 (b)) also can reduce the NoC traffic. The impact of NoC on the performance is emphasized especially for the layer with dense connections. Fig. 4.11 (b) shows that there is no throughput degradation from the locally connected layer to the fully connected layer since there is no lateral traffic on the NoC. However, one single router needs 17 input/output channels. High radix NoCs may be an option here.

4.5 Hardware Simulation

Design of PE: One PE was designed in Verilog and synthesized using 28nm CMOS and 15nm FinFet [100, 101]. The PE includes 16 MAC units, PNG, temporal buffer, and weight memory. As already explained, 2.5KB local memory is composed of 16 banks and each bank is designed to store 1,280bits (80bit $\times$ 16 lines) : 20bit word (16bit data + 4bit MAC-ID) $\times$ 16 MACs $\times$ 4 buffering depth. For each vault, I design one PE and a router for 2D mesh. The maximum clock frequency of SRAM from the 28nm library was 300MHz. Therefore, the PE and the router were synthesized to operate at 300 MHz and the MACs are synthesized to operate at 18.75MHz (Eq. 4.1). I have also re-designed the NeuroCubewith 15nm FinFet process [101] to achieve 5GHz operating frequency. Power and area of SRAM in 15nm is estimated using [102] (power). Supply voltage ratio is used to estimate SRAM power in 15nm, which is conservative estimation.

Power estimation of HMC: The baseline HMC design reports 3.7pJ/bit for DRAM and 6.78pJ/bit for logic layer [60]. Based on these values, the power of the logic die without NeuroCube (16 vault controllers, 4 links (SERDES), interface between all VCs and all links) and DRAM die is computed assuming clock frequency of the vault I/O clock (2.5GHz $\times$ 2 = 5GHz) [i.e. Logic power = 6.78pj/bit $\times$ 32 bit $\times$ 16 $\times$ 5GHz = 17.3 W]. However, the maximum clock frequency for the PE in the 28nm node is only 300MHz,

leading to a reduced activity of 0.06 (=300MHz/5GHz) in the vault controllers and DRAM. Hence, logic die and DRAM powers in 28nm are scaled accordingly. As the PE in 15nm operates at 5GHz, no such activity scaling has been applied. However, the baseline power of the logic die have been scaled based on the energy scaling factors from [103].

System power and performance: Table 4.1 shows the dynamic power consumption and area for NeuroCube in 28nm and 15nm nodes. Therefore, additional power overhead due to the NeuroCube on the logic die is 249mW (16 $\times$ 15.6mW) in 28nm and 3.41W in 15nm. The image processing throughput for inference using NeuroCubeis mentioned in Section 4.4.

Area analysis: The area overhead in the logic die due to 16 PEs is 3.09mm^2 (16 $\times$ 0.1936mm^2) in 28nm and 0.98mm^2 in 15nm. To estimate total logic die area with NeuroCube, I present one feasible layout in 28nm (Fig. 4.12). One PE and a router can be placed in 513μm^2 by 513μm^2 with 70% utilization ratio. I used area of the VC synthesized in 28nm from [62]. As there are 1,866 TSVs in one HMC [60], I assumed that 116 TSVs are placed in the middle of the VC and the area of the TSV array is estimated using a 4μm pitch and 2μm diameter [103]. One core of NeuroCube (a PE, a router, and a VC) is designed by placing VC in the middle and placing other modules around the VC to reduce interconnect. I can see that NeuroCube with 16 cores can be synthesized on the logic die (68mm^2 [60]) of HMC. NeuroCubein 15nm also fits in the area of HMC (Table 4.1).

Thermal analysis: For thermal analysis of NeuroCube ,I use [104, 105] and simulate the floorplan shown in Fig. 4.12 assuming passive heat sink. For the 28nm node, the thermal effect was negligible as NeuroCube consumes relatively small power at 300MHz (1.3W). For the 15nm node (and associated power density), I observe that the maximum temperature for 16 PEs increases up to 349K and the maximum temperature for 4 DRAM dies increases to 344K (Fig. 4.12). According to the HMC 2.0 [106], the maximum operating temperature of logic die is 383K and that of DRAM die is 378K. Therefore, NeuroCube operating at 5GHz at the 15nm node fits within thermal conditions.

I estimate logic die power based on [60], which includes the power of 16 vault controllers, 4 high speed links, ECC, and the interface between vaults and links. Note in NeuroCube , I will use links, ECC, and interface only during programming by host, not during computation; this may reduce the logic die power from Table 4.1.

4.6 Conclusion

This chapter presents the NeuroCube - a programmable and scalable digital architecture for neuro-inspired algorithms. The NeuroCube design incorporates a logic die with an array of clustered, data-driven, multiply-accumulate (MAC) units integrated with a 3D DRAM stack for high bandwidth and low latency memory access. Programmable address sequence generators integrated into the memory system generate the correct sequence of data accesses to push data from memory to the MAC units where the arrival of neuron states and connectivity weights triggers MAC operation. By reprogramming the sequence generators, multiple, different types of neural networks can be emulated. Next steps involve scaling this implementation across multiple cubes to support much larger networks than can be feasibly supported today.

Table 4.1: Hardware simulation of single core in NeuroCube

		Operating Freq (MHz)		Dynamic Power (W)		Area (mm^2)		Power Density (W/mm^2)	
	Size (bit)	28nm	15nm	28nm	15nm	28nm	15nm	28nm	15nm
MAC	16	18.75	320	3.02E-04	9.17E-03	0.0011	0.0002	2.75E-01	4.89E+01
SRAM Cache (2.5KB)	20,480	300	5,120	2.93E-03	2.90E-02	0.0873	0.0448	3.36E-02	7.10E-02
Temporal Buffer	512	300	5,120	2.70E-05	2.05E-05	0.0025	0.0003	1.09E-02	7.10E-02
PMC	N/A	300	5,120	4.17E-04	1.39E-03	0.0081	0.0013	5.16E-02	1.09E+00
Weight Reg	3,600	300	5,120	1.84E-04	1.44E-04	0.0173	0.0020	1.07E-02	7.10E-02
Router	36	300	5,120	7.17E-03	3.59E-02	0.0609	0.0085	1.18E-01	4.24E+00
PE Sum	-	300	5,120	1.56E-02	2.13E-01	0.1936	0.0600	8.04E-02	3.06E+00
Compute in NeuroCube (16 PEs + 16 Routers)	-	300	5,120	2.49E-01	3.41E+00	3.0983	0.9601	8.04E-02	3.06E+00
HMC Logic Die Without NeuroCube	-	300	5,120	1.04E+00	8.67E+00	-	-	-	-
All DRAM Dies	-	300	5,120	5.68E-01	9.47E+00	-	-	-	-

CHAPTER 5

HYBRID DATA FLOW OF NEUROCUBE FOR GLOBAL CONNECTIONS

From the Chapter 4, processor in memory architecture for deep learning application is introduced. It shows system throughput about 130GOPs/s for convolution layer with *boundary duplication* and fully connected layer with *state duplication* (Fig. 4.10).

However, relying on statically known mappings by itself becomes impractical for large dense networks (e,g. fully connected layers) with batching or large input data since it can lead to excessive data transfers between vaults and PEs. Congestion in the 2D network in the logic layer becomes the bottleneck.

For example, consider a DNN mapped to a HMC where neuron states and weights are spread across the vaults (see Fig. 5.1). Now consider the evaluation of the state of a single neuron (y_1) by PE [0]. It may need to require data from Vault 1 through NoC. This will place demands on the 2D bandwidth of the logic layer leading to congestion in the logic layer's on-chip inter-PE network. Variable delays in the on-chip network due to traffic congestion will also lead to variable arrival times of weights at a PE increasing PE idle time and reducing utilization. Conversely, from the perspective of the memory controller of a single vault, multiple requests for accesses to weights within the vault will exhibit little spatial locality. Even with optimized data mapping, the arrival times of requests from multiple PEs will differ, and thus not make use of spatial locality in mapping of weights into the vaults. The result is poorer utilization of the enormous 3D memory bandwidth (Fig. 5.1).

Consequently, to effectively harness the 3D memory bandwidth, combination of optimized data mapping and memory access scheduling are required to maximize performance.

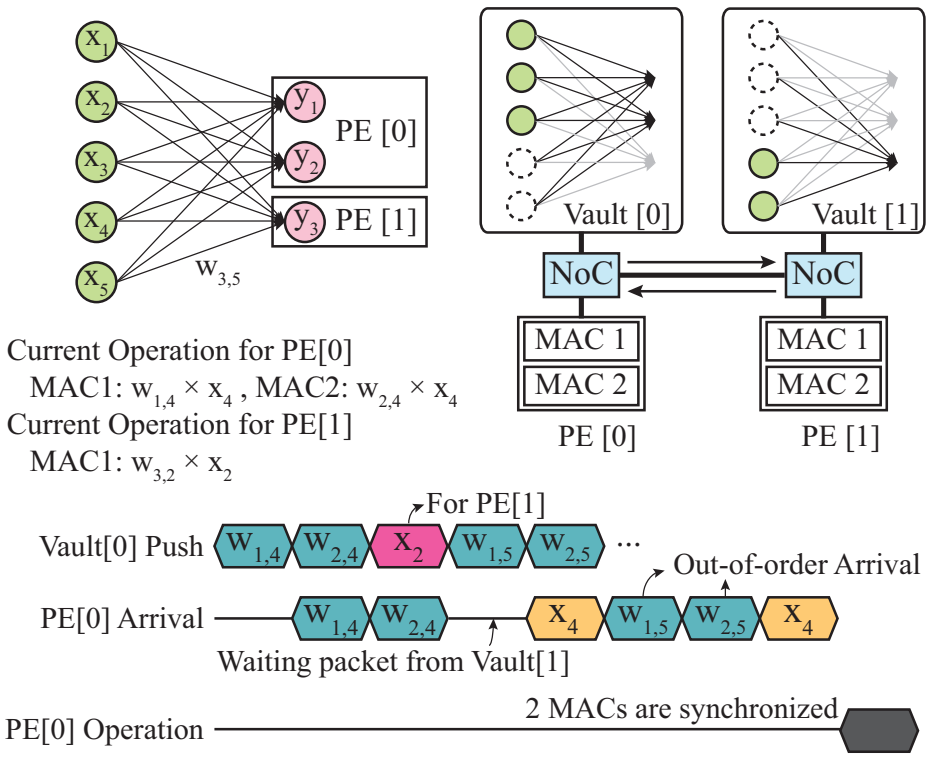


Figure 5.1: Limitation of data oriented system for fully connected layer without input duplication. PE needs data from vault through NoC (NoC lateral traffic) and out-of-order packet arrival decrease PE utilization.

5.1 Proposed Architecture For Improving Global Connection

One vault in HMC is assigned for storing states, and the remaining vaults for storing the weights. During execution, the vault (state memory) storing neurons' states broadcasts the states to all PEs, each of which is connected to a vault (weight memory). Thus weights are accessed with no inter-PE communication but rather at the full bandwidth of a vault. I consider a broadcast bus connecting the state memory to all PEs for data transfer. The architectural and algorithm parameters of the design are described in Table 5.1 where the number of processing elements (n_P) is determined by the number of memory channels (n_V) as $n_P = n_V - 1$ (n_P = 15, n_V = 16 in Fig. 5.2).

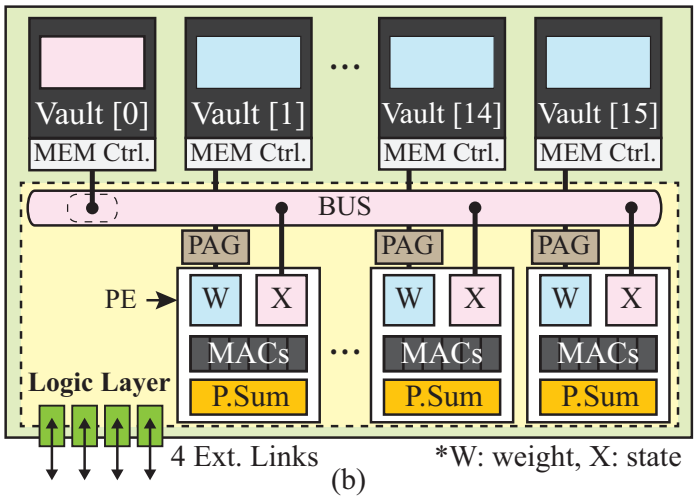


Figure 5.2: Architecture block diagram: one vault is assigned for neurons' state and connected to all PEs through BUS interface. Other vaults directly connect to a dedicated PE through high speed TSVs. A PE is composed of two cache memories (W and X), MAC units, and a partial sum memory.

5.1.1 Processing Elements (PE)

Figure 5.3 illustrates the architecture of PE[i] paired with vault[i] and connected to the bus. The key elements are state cache memory, weight cache memory, k MAC units, and partial sums memory.

State and Synaptic Weights Cache Memory

As shown in Figure 5.3, by providing separate interface paths to the vault and broadcast bus, the state and weight cache memories can be filled concurrently. Both caches are double buffered and DRAM data streams are terminated with an END-MARK (0xFFFF) symbol to simplify the cache-DRAM interface. When both caches are filled, the PE operation is triggered and the two caches *push* data into the MACs.

SIMD MAC Arrays

As the primitive arithmetic operation in DNNs is weighted summation, the PE is comprised of k multiplier and accumulator (MAC) units. Recent work has shown that 16 bit fixed point

Table 5.1: Parameters for the proposed architecture.

Notations	Meaning
n	# neurons in current layer
m	# neurons in previous(current) layer
k	# MACs in PE (mini batch size)
h	# partial sum memory depth
d	Length of partial input (X)
n_P	# PEs
n_V	# Memory channels
NZs	1D Array of non-zero values in matrix
CID	Column index of non-zero values in matrix
RID	Row index of non-zero values in matrix

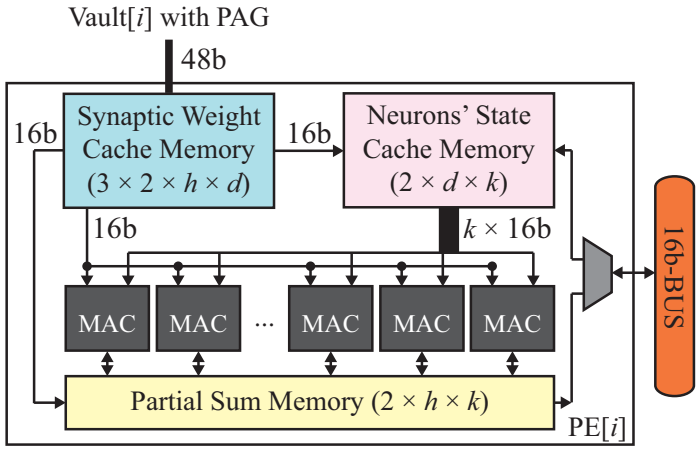


Figure 5.3: Architecture of PE[i] composed of synaptic weights cache memory, neurons' state cache memory, partial sum memory, and MACs.

precision is sufficient for inference in deep learning [27, 28]. In this design, a single MAC can compute two pairs of 16 bit operands ($Q_{1,3,12}$: 1bit:sign, 3bit:integer, 12bit: fractional value) in a single clock cycle. In other words, $2 \times k$ MAC operations can be computed in every PE cycle.

5.1.2 Programmable Address Generator: PAG

Programmab Address Generator (PAG) is similar to programmable neurosequence generator (PNG) in Chapter 4 in terms of address generating. Different connectivity between layers (convolution, recurrent, or fully connected) is accommodated by appropriate pro-

gramming of the PAG. When m neurons in layer l are connected to n neurons in layer $l+1$, the interlayer connections (weights) are modeled in an (n by m) matrix, each weight vault stores (partial) weight matrix (n and m), and each PE performs (m by k) multiplications for each neuron in layer $l+1$. During the write-back, PAG in the state vault also generates the address for states in layer $l+1$. The state vault PAG also applies the non-linear activation function using a look-up table (LUT). For higher accuracy, the LUT is designed to store both $f(x)$ and $f'(x)$ (Taylor expansion). Note PAGs in the weight storing vaults do not need to apply the non-linear activation function.

5.2 Programming and Execution

Programming NeuroCube is same as that of NeuroCube: layer-wise programming by host before operating single layer. Initially, the host loads the data into the memory vaults, configures the PAGs based on the layer type, and initiates the operation of accelerator. Once a layer completes execution, the accelerator signals the host.

5.2.1 Data Mapping for Programming

In offloading the computation of a layer, the host follows a pre-processing and mapping step to determine (i) how to distribute the weights and states across the vaults, and (ii) how to store the weights in a vault to support sparse connectivity. The **distribution of weights and states across vaults** is driven by the considerations for dense networks and support for batch processing. Fig. 5.4 illustrates how batch processing transforms matrix vector multiplication to matrix matrix multiplication $W \times X = Y$, by concatenating k input vectors as shown (Fig. 5.4). The number of vectors (k = 7 in Fig. 5.4) processed in parallel is determined by the number of MAC units in a PE. A single memory vault is assigned to store neurons' state which is broadcast to all PEs (Fig. 5.4 (a)). To process matrix-matrix multiplication in parallel, the weights matrix is divided into n_P vaults horizontally (n_P = 4 in Fig. 5.4).

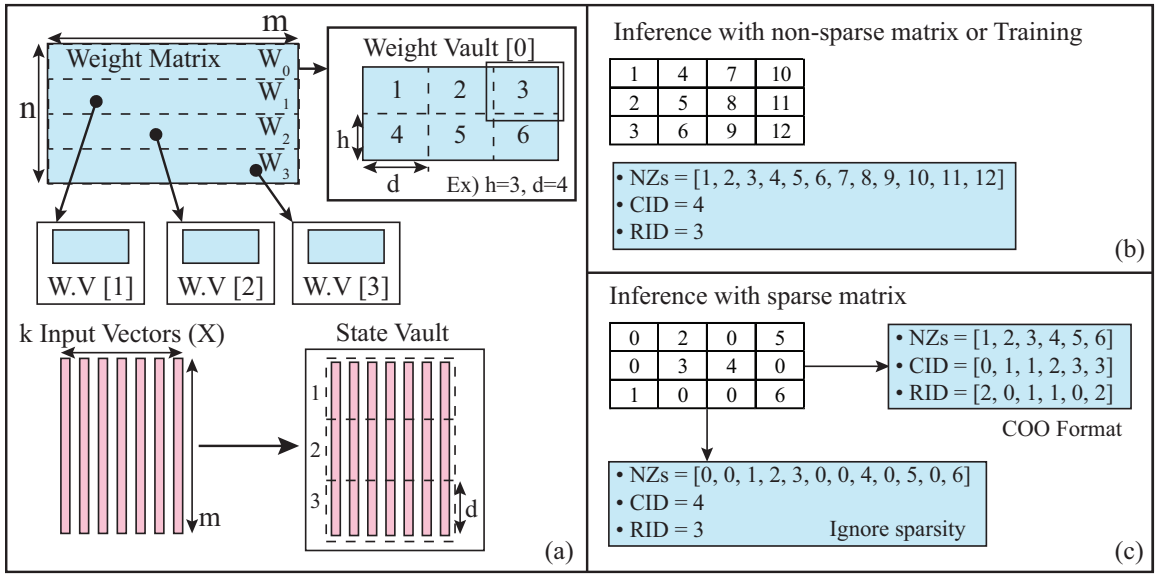


Figure 5.4: Data pre-processing by the host. (a) synaptic weights matrix partitioned into $n_V - 1$ vaults and k input vectors are stored in a single state vault, (b) if sub block of weight matrix is non-sparse or operation is training, 2D sub block is converted to 1D array without any modification and storing number of rows and number of columns, (c) if sub block of weight matrix is sparse enough (more than $\sim$60% of elements are zero), it is converted to COO format. Otherwise, it is regarded as a non-sparse matrix.

A second key decision is **how to store the weight matrices within a vault**. Within a vault, a partial weight matrix (W_0, W_1, etc.) is divided into multiple sub-blocks and each sub-block size (h by d) is determined by the size of the PE's weight cache. As an example, Fig. 5.4 shows a weight vault with six sub-blocks. If a sub-block of a weight matrix is sparse, it is encoded using the Coordinate list format (COO) into three arrays: data (NZs), column id (CID), and row id (RID). If sparsity (percentage of zeros) is low, the size of the encoded sub-block (18 in Fig. 5.4) could be larger than the un-encoded sub-block (14 in Fig. 5.4). I empirically established I should convert a sub-block to COO format when sparsity is lower than $\sim$0.4; otherwise handle the sub-block as non-sparse (Fig. 5.4 (b)).

For each layer of the DNN, the programmable address generator (PAG) should be re-programmed to determine the range and sequence of sub-blocks that should be pushed to the PE from the corresponding vault. For example, in Fig. 5.4, the PAG for the weight vault should be programmed to send 2×3 weight sub-blocks in order. Therefore, two nested

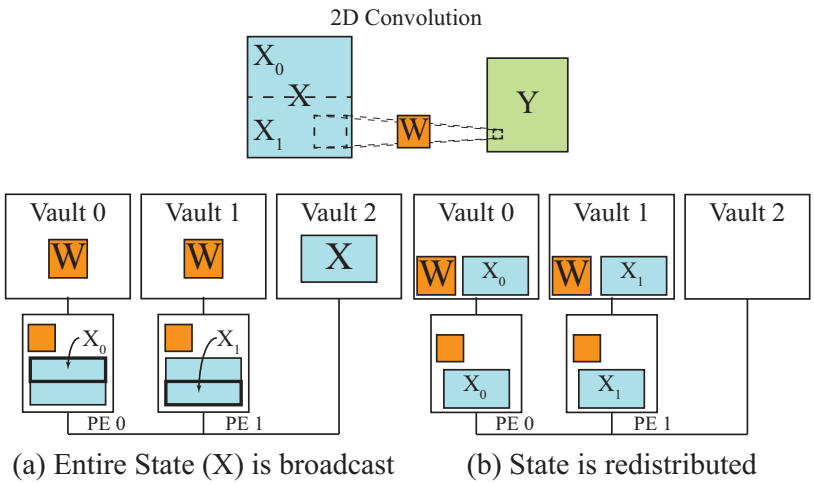


Figure 5.5: Hybrid data mapping scheme. (a) State broadcast data mapping and (b) State distribution mapping for 2D convolutional layers.

counters in the PAG will generate addresses for the sequence of six sub-blocks (1,1), (1,2), ..., (2,3). The PAG for the state vault is also programmed to send 3 state sub-blocks twice - since in this instance two rows of sub-blocks in the weight matrix are being processed. For example, in Fig. 5.4, two nested counters in PAG will generate the sequence : (1,2,3,1,2,3).

Dynamic Data Mapping: The state broadcast based data mapping does not maximize performance of locally connected 2D convolutional layers; rather partitioning *state* into multiple 2D blocks and distributing across vaults (state distribution based mapping), as proposed in Chapter 4, is effective for 2D convolutional layers. The estimated throughput (considering 15 vaults and PEs) for convolutional operations shows potential of significant performance gain by using the state distribution data mapping (over the state broadcasting) without any change in the architecture (Fig. 5.5). Therefore, I propose a dynamic data mapping scheme where convolutional layers operate with state distribution based mapping while global networks (e.g. RNN layers) operate with state broadcast based mapping. The challenge for dynamic data mapping is the overhead of re-programming the mapping between convolutional and fully connected layers. However, such re-programming will be rare as for example in a typical DNN, a series of 2D-convolutional layers is followed by a series of dense layers.

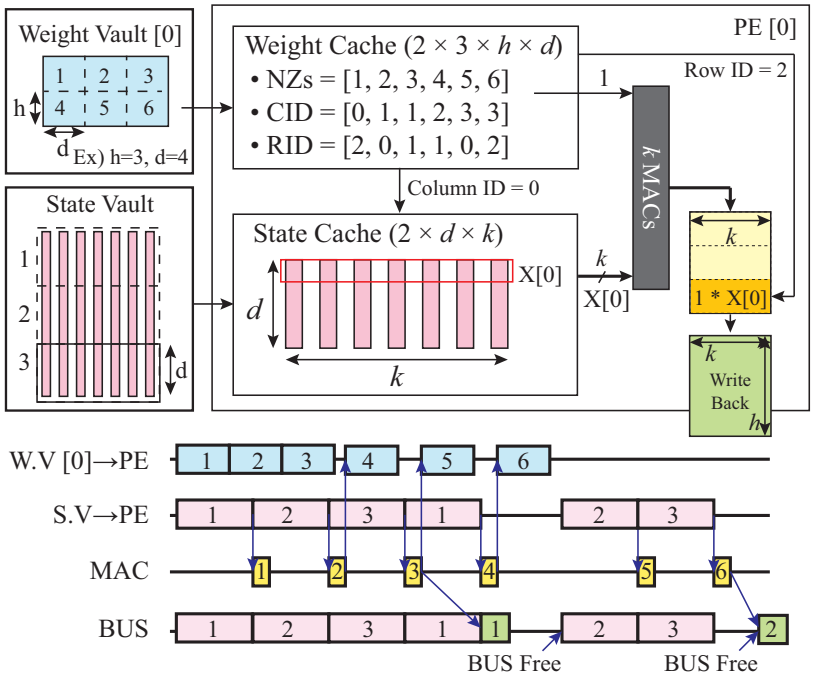


Figure 5.6: Data flow in the operation of a PE.

5.2.2 Execution Flow

After storing the synaptic weights and states into assigned vaults, the vault controllers with the PAGs start to access and push data to the dedicated caches as I explained in Section 5.2.1. When both sub-blocks are ready in cache memory, the PE starts a MAC operation. The weight cache pushes first items in three arrays (NZs = 1, CID = 0, RID = 2 in Fig. 5.6). The CID is delivered to the state cache as a row address, then the state cache will push a k-length row of state (X[0] in Fig. 5.6). Both X[0] and NZs are delivered to k MAC units and the output is delivered to a partial sum memory. The RID is delivered to partial sum memory as the write address (P[2] in Fig. 5.6). For a non-sparse weight block, k MACs needs $h \times d$ cycles to finish one block (one yellow block in Fig. 5.6). For a sparse weight block, the number of cycles to finish one block is the same as the number of non-zero weights (6 in Fig. 5.6).

Since cache capacity is double buffered, half of the cache can be filled while the other half can deliver data to the MACs. Fig. 5.6 shows the data flow for PE[0]. For example,

Table 5.2: Energy consumption per operation for each module in 15nm FinFet process [101].

Module	Target Freq. (GHZ)	pJ/Ops
16b MAC	2.50	1.69
16 KB SRAM (64B word)	3.69	13.3
HMC DRAM Cell [60]	2.50	118.4
HMC Vault Ctrl. [66]	2.50	1.7
BUS Interface	2.50	1.4

when the weight vault finishes sending the 2nd block, the 1st block can be flushed (already used for 1st computation); therefore the 3rd block can be delivered while keeping the 2nd weight block in the cache. In general, transfers between the weight vault and PEs, and the PE operation can be overlapped. The weight vault and PEs use an asynchronous handshake protocol for transfers to even out variations in computation time and simplify scheduling of the vault-PE-vault data flow.

For example, after k MACs finish computing with the 3rd block, the value in the partial sum memory (Y_1 in Fig. 5.6) is delivered to the state vault (write back) over the bus. Since multiple PEs share a single bus, during the multi-PEs write back, the state vault cannot broadcast state to PEs. However, the amount of data for writing back (h by d) is smaller than that of the state (d by k) or that of weights (h by d). Further, the write back occurs only after finishing with a weight block in a row (1,2,3 in Fig. 5.6). Therefore, I estimate that the performance overhead due to write back is not significant.

5.3 System Performance

Proposed architecture is modeled in System-C including DRAM, bus interface, address generator, and PEs. Except DRAM, all modules are synthesized with 15nm FinFet technologies [101]. As illustrated in Fig. 5.4, PE utilization is low due to data movement from DRAM to PEs. To estimate energy consumption, I use estimates of the energy of each modules as shown in Table 5.2 assuming a switching activity factor of 0.3. For SRAM, I

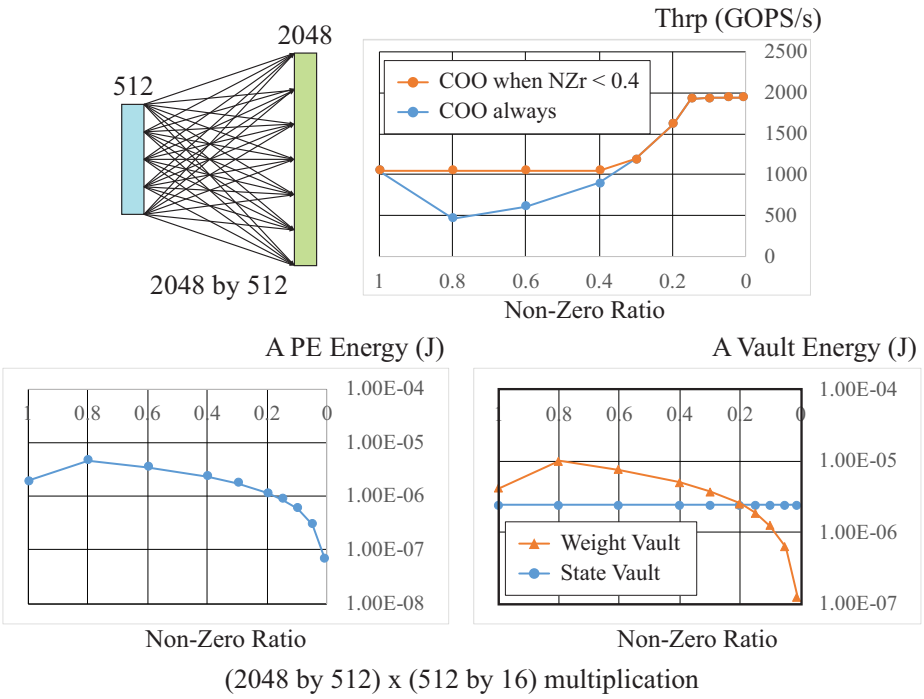


Figure 5.7: Cycle-level simulation for throughput (GOPS/s), single PE energy (J), vault energy (state/weight) (J) for different non-zero ratios (0.8: 80% of synaptic weights are non-zero).

use average access energy across read and write operations.

5.3.1 Impact of Sparsity

Fig. 5.7 shows throughput (GOPS/s), energy of a single PE (16 MAC units, two cache memories), and energy of a single vault (state vault or weight vault) for different non-zero ratios (NZr) (ex: NZr = 0.8 means 80% of synaptic weights are non-zero) for dense connections from 512 neurons to 2,048 neurons (mini-batch size = 16). As explained in Section 5.2.1, when sparsity is low, the number of transfers considering sparsity (COO encoded arrays) is higher than transfer without considering sparsity. That is why system throughput decreases when NZr is high and recovers when NZr is about 0.4. In terms of PE energy consumption, sparse synaptic connection can reduce both weight cache memory energy and MAC array energy consumption. In the same way, the weight vault's energy consumption (average data access energy - both read and write [60]) is also reduced while

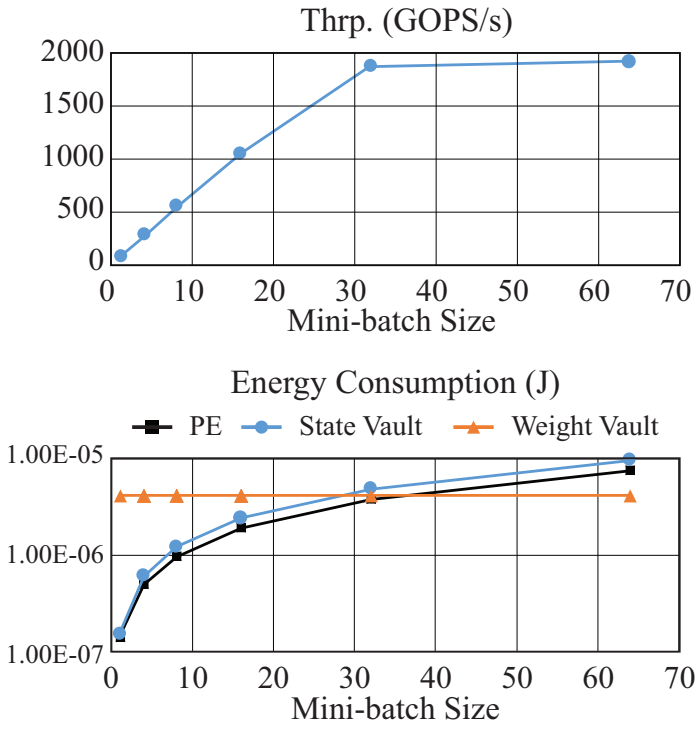


Figure 5.8: Impact of mini-batch size on system performance: 2048-to-512 connection, NZr = 1.

state vault's energy is constant (no change in terms of number of access).

5.3.2 Impact of Mini-Batch Size

To improve system throughput by re-using synaptic weights for multiple inputs, the accelerator supports batch processing. Fig. 5.8 shows system throughput and energy consumption for dense connections (NZr = 1) between 2,048 neurons and 512 neurons with different batch sizes (k = mini batch size = number of MACs in a single PE). It shows throughput is proportional to batch size until mini batch size becomes a 32. As mini-batch size increases (state cache size increases as well) beyond 32, filling the state cache in a PE becomes a bottleneck for the entire system. Therefore, although PE energy consumption and state vault energy consumption increases linearly, system throughput becomes saturated.

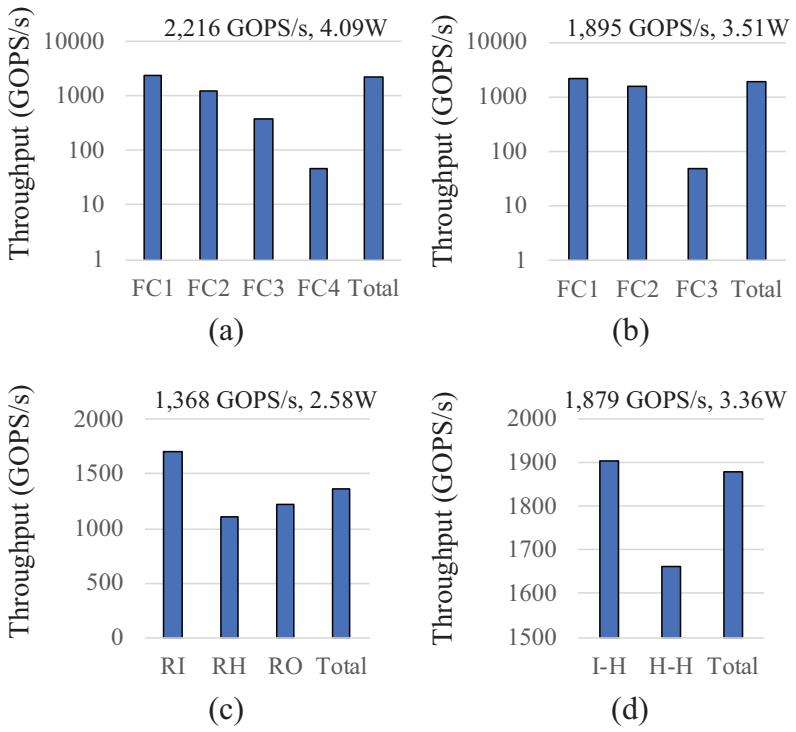


Figure 5.9: Simulation results: (a) MLP0 in [33], (b) MLP1 in [33], (c) GRU for natural language processing [40] and (d) LSTM in [33].

5.3.3 Benchmark Analysis

Dense Connections

First, real benchmarks mainly based on dense connections (MLP, LSTM, GRU) are simulated (Fig. 5.9): (a) gated recurrent unit (GRU) [40] for natural language processing with 6 dense connections between 500 input neurons and 1,500 hidden neurons, (b) and (c) MLP illustrated in [33], (d) LSTM illustrated in [33]. Since exact networks of MLP (b,c) and LSTM (d) are not provided in [33], I estimate network structures to match with information provided in [33]. To maximize the throughput, mini-batch size is fixed as 32. In dense connections, our architecture shows higher system throughput (1,300GOPS/s $\sim$ 2,200 GOPS/s) while consuming 3 $\sim$ 4 Watt including DRAM power. For four benchmarks, it shows very high average power efficiency of $\sim$530GOPS/s/W.

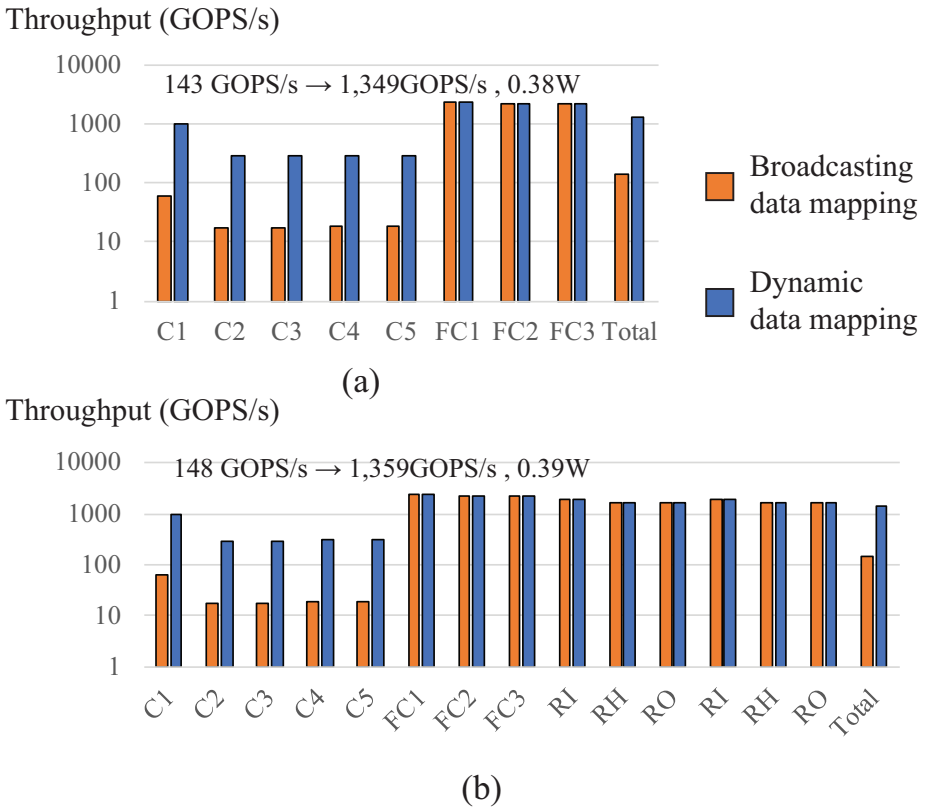


Figure 5.10: Simulation results: (a) AlexNet [1], (b) Hybrid NN (Conv-RNN) [42] for different data mapping schemes.

Hybrid Networks

Fig. 5.10 shows simulation results considering state broadcast based data mapping for benchmarks including locally connected convolution operations: (a) AlexNet [1] for image processing, (b) Hybrid NN (Conv-RNN) for sentence generation to describe an image [42] (Fig. 5.11). Similar to MLP, LSTM in [33], the Hybrid-NN architecture is estimated by combining AlexNet [1] and GRU [40]. Since broadcasting states to all PEs is not efficient in a 2D-Conv layer due to its local connectivity, the convolution layers show low system throughput (∼100GOPS). Therefore, power efficiency is also low (∼380GOPS/W) compared to dense connections. The dynamic data mapping scheme can be used to program state distribution based data mapping for convolutional layers, thereby significantly enhancing system performance for the network. The overhead of changing the data mapping

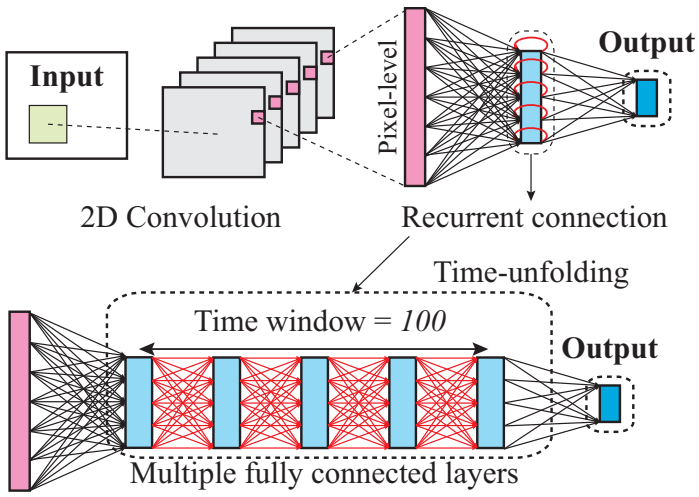
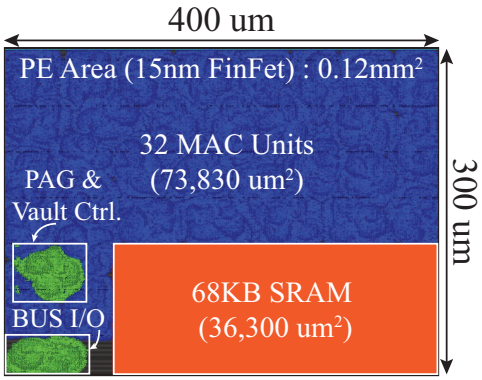


Figure 5.11: Simplified example of DNN for generating sentences for image description [42].



Module	um^2
32 MACs	73,830
State Cache (16KB)	8,540
Weight Cache (48KB)	25,620
P. Sum Cache (4KB)	2,140
PAG + Vault Ctrl.	1,967
BUS I/O Unit	2,501
Total	**120,000**

Figure 5.12: Layout of a single PE in 15nm FinFet process - area is $0.12mm^2$ with 70% utilization ratio.

between convolutional and FC layer is estimated. In AlexNet, the output of the last convolution layer is 43,264 states (86KB). To move all states from 15 vaults to a single vault using the bus interface (10 GBytes/sec), will take $\sim 8.6\mu S$, which is negligible compared to the entire latency (0.11 second when mini-batch size is 32). After re-mapping data, the throughput for the convolution layers is improved by 16X.

5.3.4 Physical Implementation

A PE including 32 MAC units, a 16KB state cache memory, a 48KB weight cache memory, a 4KB partial sum memory, BUS interface module (handshaking), address generator

(PAG), and vault controller (SDRAM controller [66]) is synthesized using 15nm FinFet. All clock frequencies for SRAM, MAC, and controllers are 2.5GHz to utilize HMC internal bandwidth maximally (32-bit TSV bus, 10 Gbyte/s). 32 MAC units occupy about 70% of area in a single PE ($0.074mm^2$). For 15 PEs, total area is $1.80mm^2$. Therefore, I expect that this design can be accommodated within the logic die footprint of a HMC ($68mm^2$) [60].

Average power (0.3 switching activity) of the logic layer (15 PEs and one bus) is 1.605W ($0.02W/mm^2$) while all 4 DRAM dies consume 1.884W ($0.03W/mm^2$). According to [68], power density below $1.5W/mm^2$ will not generate thermal issues with passive cooling in 3D stacked system; similar observations are also made in 4.

5.4 Comparative Analysis

Recent accelerator platforms for deep learning are illustrated in Table 5.3 including both programmable accelerators and customized accelerator for specific networks (mostly convolution). The prior works on in-memory accelerators are identified using **PIM** in the 3rd row (Memory) [28, 29, 30]. In terms of performance analysis, Alexnet [1], MLP [33], RNN(GRU) [40], and Hybrid NN [42] are used for this work. For the previous works in the Table 5.3, I have used the highest throughputs reported in the corresponding papers.

The throughput for customized network accelerators is limited by the external DRAM bandwidth, unless all weights are stored on-chip (e.g. eDRAM [24]) - this greatly limits the DNN size and therefore scalability of supported globally connected networks. The design presented [26] can handle sparse networks; however, it assumes the entire network can be compressed to store in the on-chip SRAM making the design less scalable. As an example of designs supporting multiple networks, DNPU has separate blocks for convolution and recurrent networks that operate independently [32]. In comparison, NeuroCube is scalable and programmable across all major network types.

The TPU utilizes a systolic PE array (fixed data flow) to accelerate matrix multiplica-

tions [33]. Although a TPU can support different types of networks by mapping them into a matrix multiplication formulation, it is difficult to handle sparse data efficiently as all PEs in a systolic implementation need to follow a common data movement. Moreover, the throughput for MLP or RNN layers drops compared to convolution as data mapping is not optimized.

The most closely related works are the in-memory accelerators that use 3D (or 2.5D) integration of logic layers with die-stacked DRAM (HMC) [28, 29, 30]. Neurostream uses a cluster of PEs and with RISC core as a controller [30]. Tetris places more than 3,000 computing units and small on-chip buffers on die to reduce power/area overhead of on-chip memory [29]. Neurocube uses a data flow architecture with a 2D array of PEs on the logic layer and a memory address generator in the vault controller [28]. These designs mainly focus on convolution layers (computing intensive) rather than fully connected layer (memory intensive) - the latter can require significantly more data movement. NeuroCube improves on these works with power efficient acceleration of RNN and hybrid networks (Table 5.3).

In summary, the NeuroCube advances the state-of-the-art in several ways. In contrast to most of the prior works, the NeuroCube demonstrates run time programmable acceleration of CNN, MLP, RNN, and hybrid networks. Further, unlike prior works, the NeuroCube is programmed by specifying the memory access patterns and data flow with support for compression, e.g., unlike the TPU. Although the Neurocube in Section 4 supports both convolutional and FC layers, it duplicates states into all memory vaults for the FC layers which is impractical for large networks [28]. In this chapter, NeuroCube enhances performance of globally connected networks by using adaptive data mapping, adaptive compression, efficient communication architecture, and in-memory computation. Prior works did not present effective data mapping and communication strategies for fully connected layers. Consequently, the NeuroCube offers similar performance for CNN, MLP, RNN, and hybrid network as well as effectively supporting sparsity. It is contrast to TPU [33],

Table 5.3: Recent Hardware Platforms for Deep Learning.

	DNN Accelerator for Specific Network(s)						**Programmable DNN Accelerator**			
Name	**Eie '16 [26]**	**DDN '14 [24]**	**Eyeriss '16 [27]**	**DNPU '17 [32]**	**RNN '16 [31]**	**Tetris '17 [29]**	**TPU '17 [33]**	**NC '16 [28]**	**NS '17 [30]**	**This Work**
Memory	SRAM	eDRAM	DRAM	DRAM	SRAM	**PIM**	DRAM	**PIM**	**PIM**	**PIM**
Tech (nm)	28	28	65	65	14	45	28	15	28	15
Bit	4 fixed	16 fixed	16 fixed	4-16 fixed	N/A	16 fixed	8 fixed	16 fixed	32 float	16 fixed
Sparse	Yes	No	No	Yes	No	No	No	No	No	Yes
Thrp. (CNN)	N/A	5580	12	265	N/A	1568	86,000	135	955	1,349
Thrp. (MLP)	448	N/A	N/A	231	N/A	N/A	12,300	126	N/A	1,879
Thrp (RNN)	188	N/A	N/A	231	124	N/A	3,700	N/A	N/A	1,580
Thrp (Hybrid)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1,359
Power (W)	2.36	15.97	N/A	$0.03^{\#}$	N/A	6.94	384.00	12.88	10.00	3.49
Efficiency (GOPS/W)	190	349	N/A	$8{,}100^{\#}$	N/A	226	224	10	95	538

DDN: DaDiannao, **NC**: Neurocube in Section 4, **NS**: NeuroStream, **PIM**: process in memory, $^{\#}$DRAM Power not included

which shows varying throughput per network - higher in CNN and lower in RNN. Although systolic arrays of PE is efficient for matrix multiplication, I can assume data movement become bottleneck of system performance; therefore RNN shows lower performance (high data movement) while CNN shows higher performance (low data movement). Unlike the DNPU where different accelerators are used for different layers, the dynamic data mapping in the NeuroCube enables similar throughput for convolution, FC, or recurrent layers using the same computing engine. Collectively, the NeuroCube demonstrates higher power-efficiency than the prior works, particularly for globally connected networks (MLP, RNN, and hybrid networks).

5.5 Conclusion

This chapter presented next version of NeuroCube for deep learning networks that employ globally and locally connected layers, including the use of temporal feedback. The data flow based computing model provides the programmability for wide classes of DNNs while optimized data mapping provides significant performance gain. As the complexity and application of DNNs continue to grow, NeuroCube can provide the scalability, flexibility, and power efficiency necessary for future DNN accelerators.

CHAPTER 6

DEEP LEARNING ACCELERATOR FOR BOTH INFERENCE AND TRAINING

As the last version of NeuroCube, it is designed to cover both inference and training for most of DNN structures (convolution layer, fully connected layer, and recurrent layer). The training of a DNN is composed of three primary steps: forward propagation (FP), which is identical to inference, back propagation (BP), and parameter update (UP).

The acceleration of training faces major additional challenges over acceleration of inference as discussed below.

1) Most DNN accelerators for inference are optimized for convolutions with small kernels and matrix-matrix multiplication (for fully connected layers). However, accelerating BP and UP includes the following additional operations - i) convolution with very large kernels, ii) matrix transpose, iii) vector to vector outer product, iv) loss function computation, v) a pooling layer and its derivative, and vi) the derivative of non-linear activation functions.

2) Training operates over very large data sets and employs mini-batch processing across training thereby requiring larger on-chip storage to increase effective memory bandwidth. In contrast, inference operates over single sample. Further, while inference requires only reading weights, training requires reading/writing weights and their gradients, increasing memory traffic.

3) The computation of gradients in back propagation and weight update require higher bit precision to account for small gradient values (vanishing gradient issue [11, 107]). Therefore, low bit recision (8/16 bit) arithmetic, often used during inference for energy efficiency, is not suitable for training.

Similar to Section 5, NeuroCube has two separate data paths and utilize it based on the layer operation. Main improvements are summarized as below:

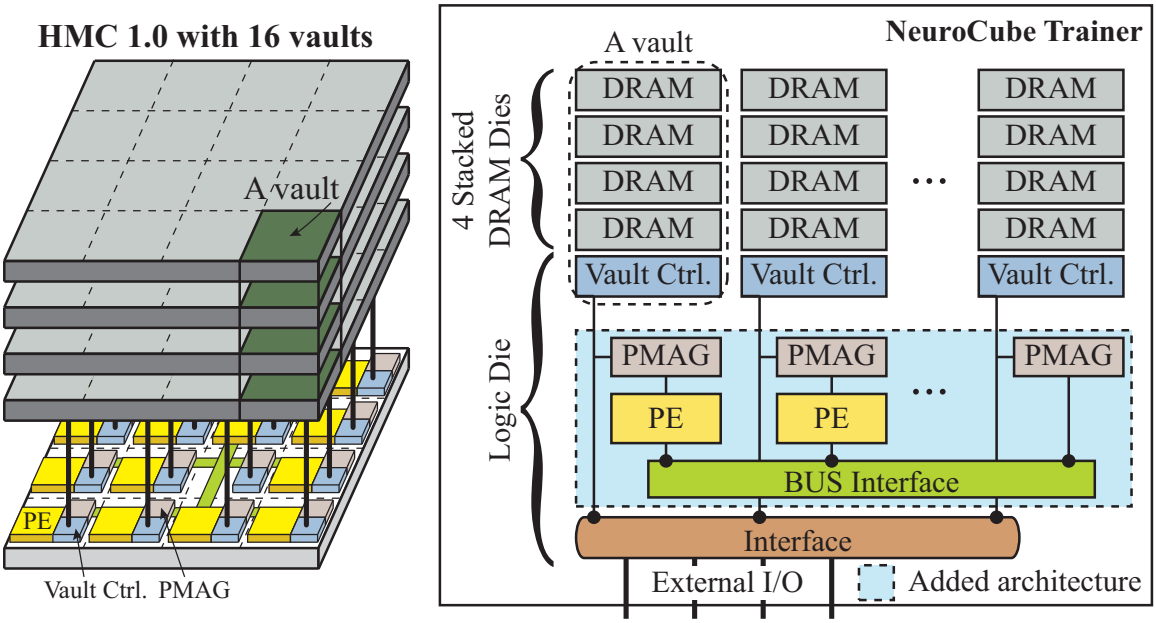


Figure 6.1: Proposed architecture as processor in memory within a Hybrid Memory Cube [58].

1) MAC design to operate 32bit fixed point with stochastic rounding and 16bit fixed point. To acheive high accuracy in training, 32bit fixed point is not enough. However, floating point unit is too heavy to use in PIM architecture especially for power density issue. MAC with two operating mode for both inference and training is used.

2) More complicated address generator design, Programmable memory address generator (PMAG) is designed with 7 level nested counters and combinational logic to cover all operations in feedforward, backpropagation, and weight update.

6.1 Proposed Architecture

There is no architectural difference from NeuroCube in Section 5 (Fig. 6.1). Each PE has a one high bandwidth connection to its local vault and an interface to a broadcast bus connected to a shared vault. The vault controllers are augmented with a programmable memory address generator (PMAG), a state-machine that realizes mapping of the different types of data (input, parameter, and gradients) to different vaults and control data flow

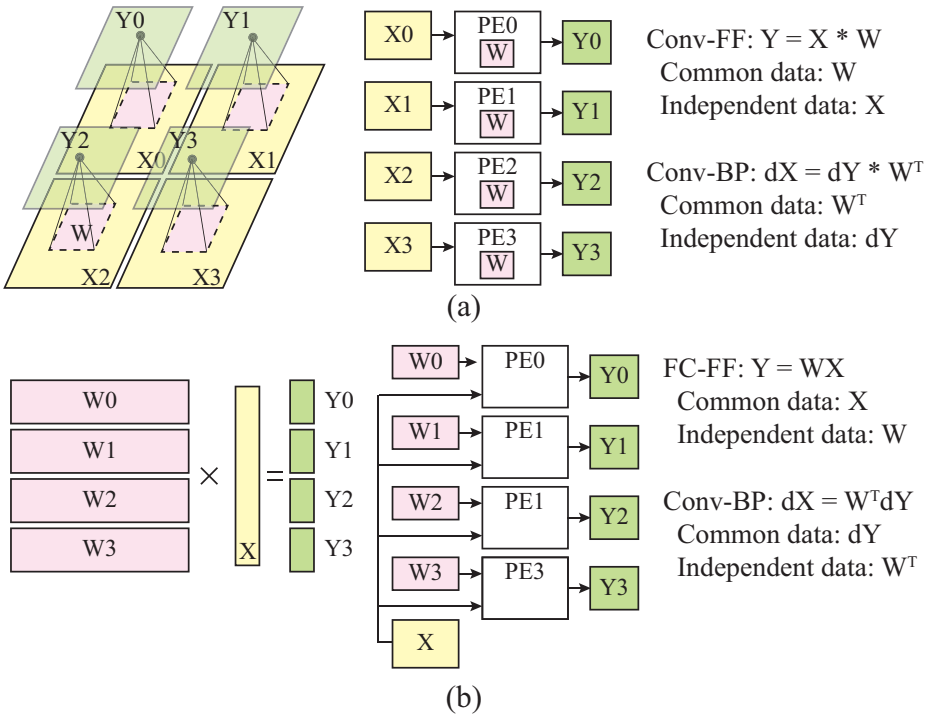


Figure 6.2: Hybrid Data Flow. (a) 2D convoltuion with small kernels by 4 PEs in parallel. Small common data (kernels: W) is pre-stored in PE's memory. (b) Matrix multiplication by 4 PEs in parallel. Large common data (X) is broadcasted to all PEs and partial weight matrix (W0 - W3) is feeded into 4 PEs in parallel.

between memory and PEs.

6.1.1 Hybrid Data Flow

Hybrid data flow introduced in Section 5 is also used for training as well, but more general fashion. Consider convolution and matrix-matrix multiplication. During the convolution (Fig. 6.2 (a)), kernel (W) is shared by PEs while input is partitioned for each PE. For the matrix-matrix multiplication, weight matrix (W) is partitioned while input (X) is shared by PEs. We note that one of the inputs can be shared by all PEs in any operations of DNN.

Based on the size of common input, operations in Fig. 2.7 can be classified as *small common data* (ex: convolution with small kernels) and *large common data* (ex: matrix-matrix multiplication). For example, the weights of small kernels in the convolution layer would be small common kernels whereas the large weight matrix in a fully connected layer

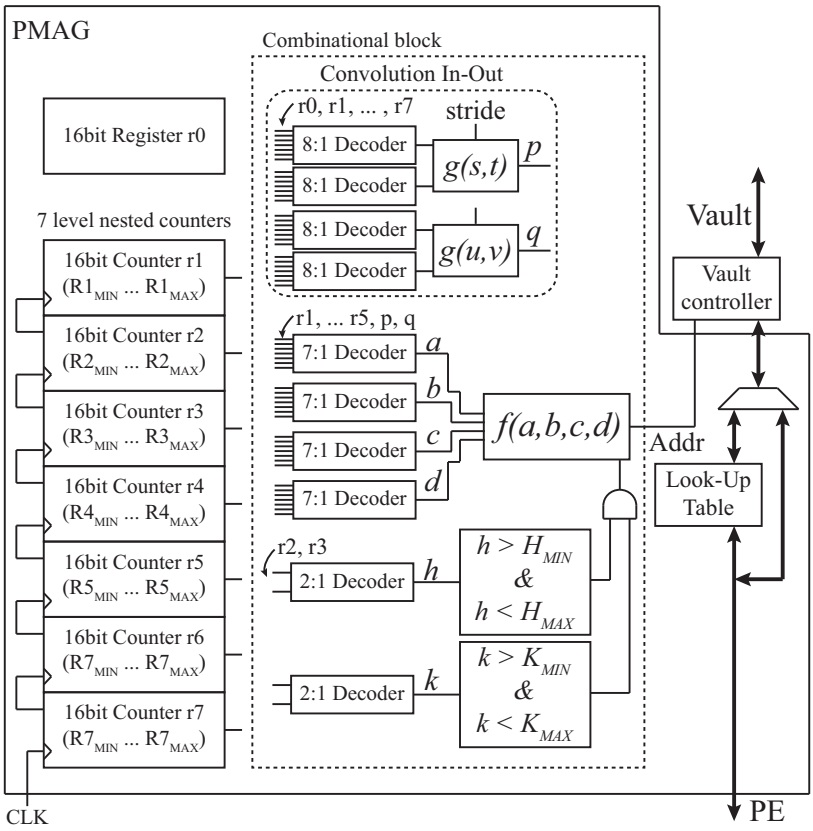


Figure 6.3: Block diagram of PMAG

corresponds to large common data. The approach used in NeuroCube is to buffer copies of small common data across PEs and stream partitions of large data (e.g. inputs to a layer) from the local vault. Alternatively, with respect to large common weight matrices, they can be broadcast from a shared vault to all PEs, while partial weight matrices are stored across PEs. These two classes of data flows are illustrated in Fig. 6.2.

The *data rearranging* among vaults is required to dynamically change data flow from one type of layer to another. However, in a DNN, a set of convolution layers is followed by a set of fully connected layers; therefore rearrange is required only once in both feedforward and backpropagation.

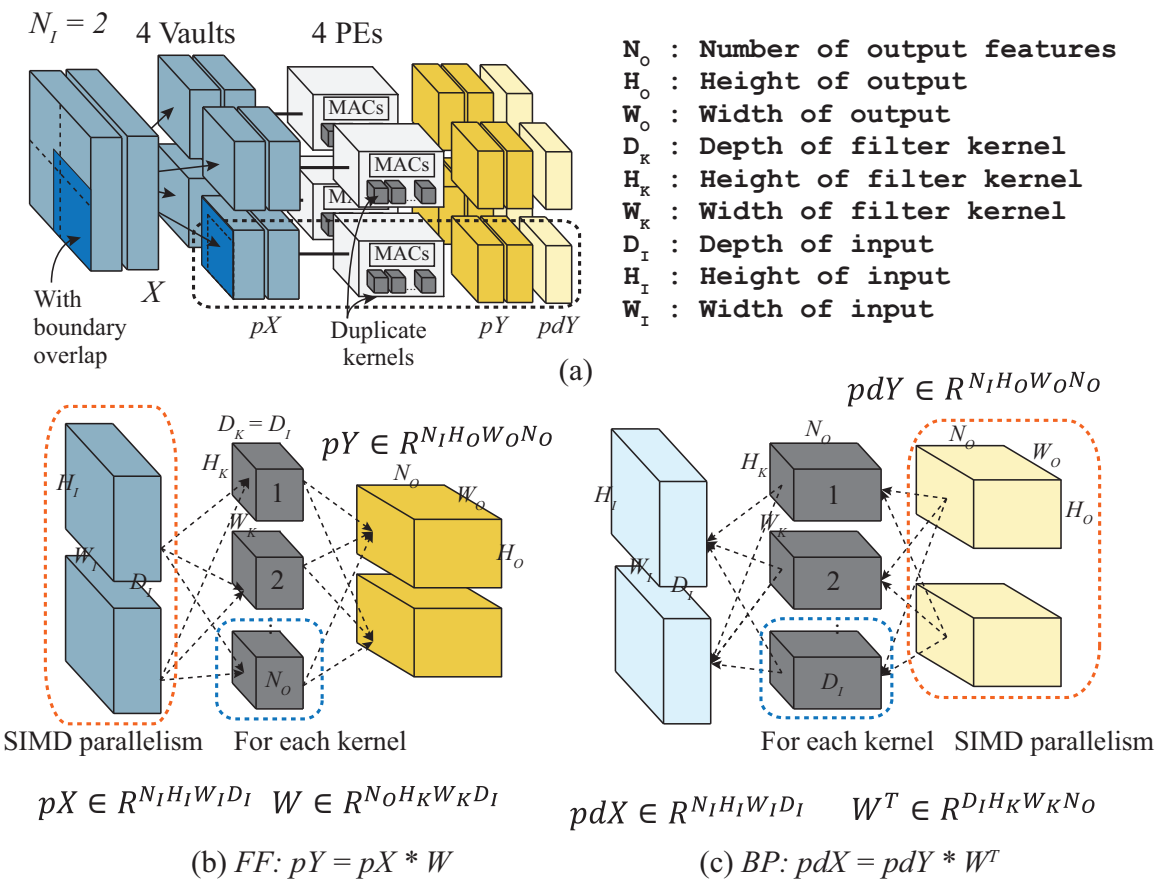


Figure 6.4: Convolution (X and W). (a) X is partitioned into 4 pX for 4 PEs, (b) Convolution feedforward, (c) Convolution backpropagation.

6.1.2 Programmable Memory Address Generator (PMAG)

The programmable memory address generator (PMAG) controls the data flow by providing memory address to the vault controller for read and write, and pushing the data through the NeuroCube. The PMAG is composed of 7-level nested counters ($r1...r7$), combinational logic to generate address, and decoders to assign counter values as input of combinational logic (Fig. 6.3). The PMAG also computes the non-linear function (and its derivative) by using look up tables [LUTs, for $f(x)$ and $f'(x)$] for (a) activation function (ReLu, tanh, etc.) or (b) exponential/logarithm for softmax and cross-entropy layer.

Convolution Feedforward / Backpropagation. Fig. 6.4 shows input X is partitioned into 4 pX with boundary overlap for convolution (assume 4 PEs). As kernel size is small,

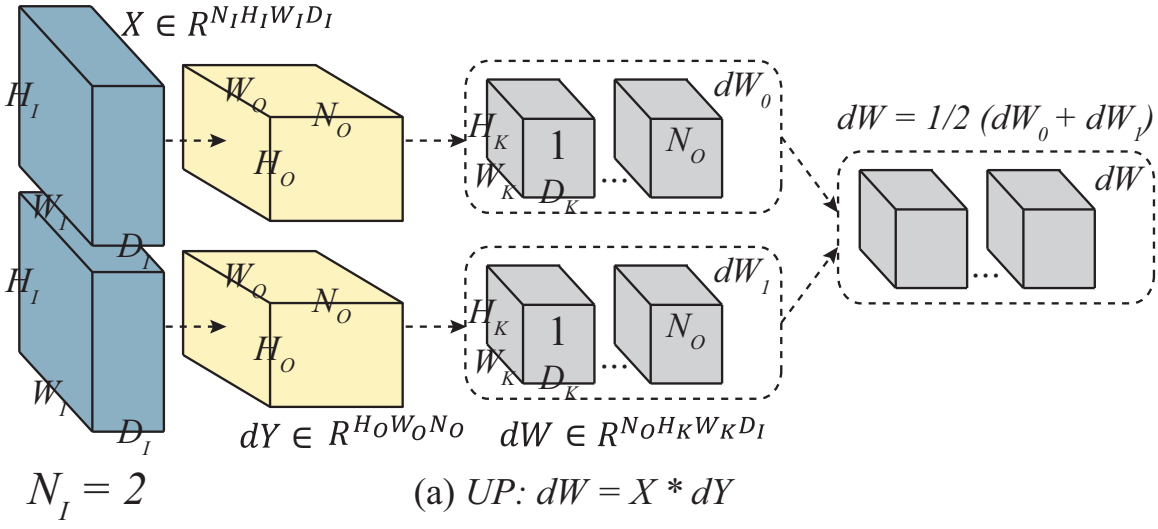


Figure 6.5: Convolution weight update when N_I is 2. It will generate dW_0 and dW_1 by $dW_i = X_i * dY_i$. Final dW is average of dW_0 and dW_1.

kernels are duplicated into all PE's buffers. For each kernel (outer most loop is N_O), N_{MAC} inputs are processed in parallel (SIMD). For backpropagation, transpose of W (W^T) is required and it can be handled in PE without reshaping data in the buffer of PE. It will be explained in Section 6.1.3.

Convolution Weight-update. After generating dY, dW is needed to update weights. Fig. 6.5 shows convolution weight update when N_I is 2. For each sample (X_i), $dW_i = X_i * dY_i$ is computed, and final dW is computed by averaging all dW_is. Although weight update is also convolution between X and dY, the kernel size (W_O by H_O) is similar to the input size (W_I by H_I). Due to large kernel (dY), partitioning input (X) with boundary overlap (Fig. 6.4) is inefficient and duplicating dY into all PEs is impractical. Therefore, we convert convolution with large kernel to matrix matrix multiplication by lowering convolution similar to how cuDNN performs convolution [108] (Fig. 6.5 (b)). Although drawback of lowering is increasing memory requirement from X_i to X_{Mi}, in-memory computation in NeuroCube can resolve the memory challenge.

Matrix-matrix multiplication. The main operation of fully connected layer or recurrent layer is matrix-matrix multiplication ($A \times X = AX$) [39, 40]. Fig. 6.6 shows that A

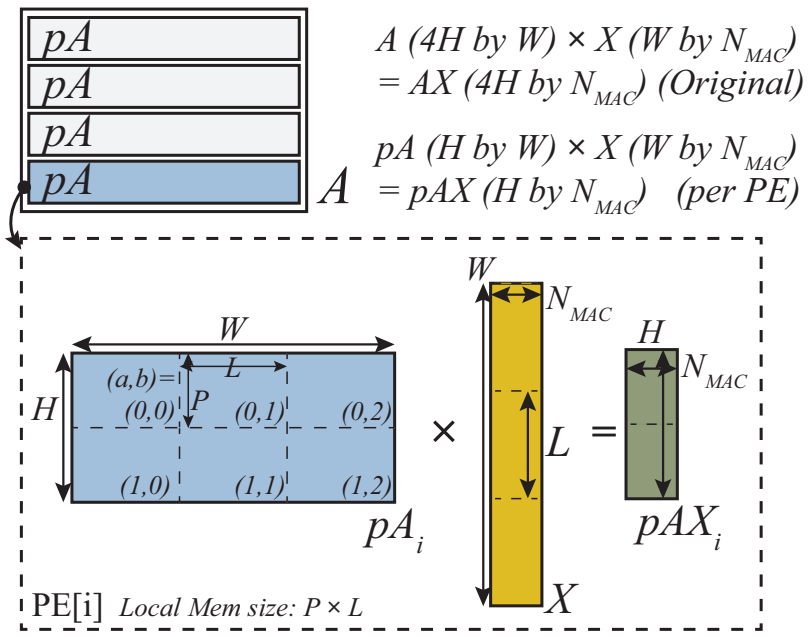


Figure 6.6: Matrix-matrix multiplication using 4 PEs. Each PE computes $pA \times X = pAX$.

is divided into 4 pA_i (i: PE index) row-wise and how a single pA_i is partitioned to small blocks (each size is $L \times P$), which is fitted into half size of buffer in PE (double buffering). As explained in Section 6.1.1, two data paths operate in matrix matrix multiplication (large common data) and the PMAG with common data vault and the PMAG with independent data vault are programmed separately. Fig. 6.6 shows that pA_i is partitioned into 3 by 2 blocks. After processing first 3 blocks of pA_i and X, a block of pAX_i is generated (size = N_{MAC} by H). The pAX_i needs to be delivered to common data vault.

Vector-Vector Outer product. For weight update in FC layer, for each sample in batch, input (X) and gradient (dY) need to be multiplied to generate dW. Contrast to matrix-matrix multiplication, N_i samples cannot be unlooped in SIMD level. In other words, this operation should be repeated N_i times and dW needs to be averaged. Fig. 6.7 shows vector-vector outer product using 4 PEs. Vector A is divided into 4 vaults (pA_i, size = H) and B will be stored in common data vault and will broadcast. The operation inside PE is similar to that of matrix matrix multiplication, however, the output ($pA'pB^T$) does not need to be merged to common vault since it's gradient of weight in FC layer; therefore

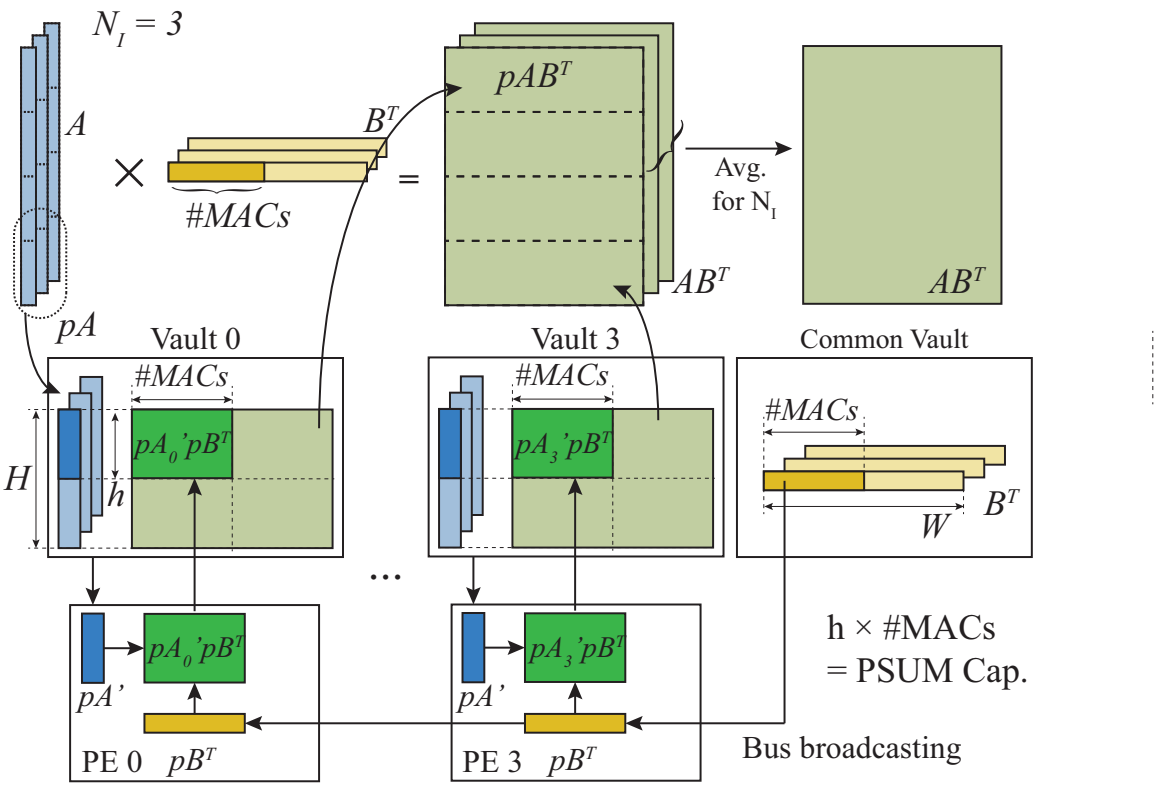


Figure 6.7: Vector-vector outer multiplication using 4 PEs. Each PE computes $pA \times B^T = pAB^T$.

it's written back to dedicated vault.

Data Preparation Fig. 6.4 (a) shows that convolution with 4 PEs generates 4 pYs in parallel. If it is the last convolution layer before fully connected layer, the outputs of convolution layer should be **merged** into common data vault before to be broadcast in matrix matrix multiplication (Fig. 6.2). The order of PE to send data is pre-determined in the BUS. Based on this order, PMAG connected to common data vault also knows the portion in the merged data (PW, PH). In similar way, data from common data vault is also **partitioned** to all other vaults.

Add/remove zero boundary. Before convolution, input needs to be **zero padded** on the boundary to return same sized output based on the kernel radius r.

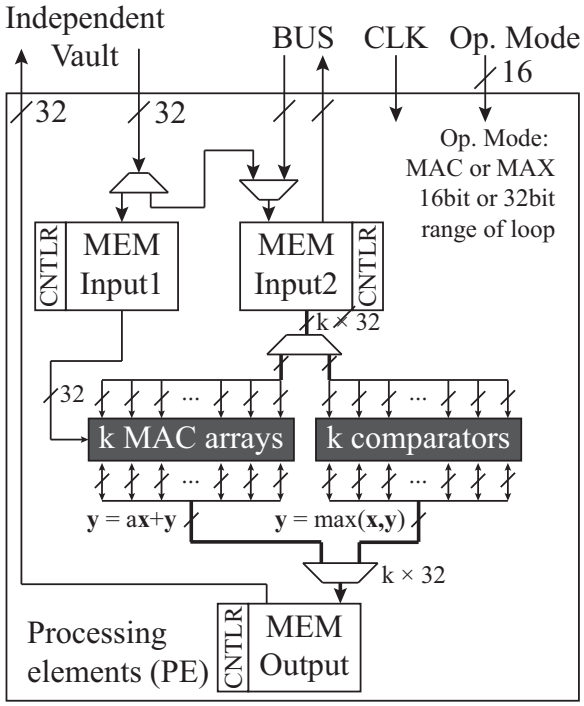


Figure 6.8: Block diagram of PE composed of three local buffers (input1, input2, and output), k MAC, k comparators.

6.1.3 Processing Elements (PE)

A processing element is composed of a k MACs array, k comparators, and three local buffers: two input buffers, one output buffer (partial sum) (Fig 6.8). Similar to PMAG, PE also needs to be programmed before the main computation.

Local Buffers

To avoid stall of PE (idle mode) due to lack of operands (inputs), we use two inputs and output buffers in PE. For each buffer, while half of memory is consumed by MACs (computing operation), rest of buffer can be filled simultaneously (double buffering). All local buffers have address generator based on nested counters. In Fig. 6.8, CNT2 is two level nested 16bit counters and CNT1 is single level 16bit counter. Data stream between DRAM and PE ends with END-MARK (0xFFFF for 16bit case, 0xFFFFFFFF for 32bit case) Computing in the PE starts only both input buffers are ready (half filled).

Table 6.1: Comparision different fixed point MAC designs with IEEE 754 single precision floating point MAC. All designs are synthesized with 15nm FinFet [101] operating at 2.5GHz

	Area (um^2)	Power (mW)
Float 32	2093.88	5.37
Fixed 32/16 + SR LO	1578.71 (-24%)	3.78 (-30%)

Multiple precision MAC Units with Stochastic Rounding

As primitive arithmetic operator, a row of k multiplier and accumulator (MAC) units is placed in a PE. As explained in Section 3.3, for training deep network, 16 or 32bit fixed point is not enough. Therefore, 32 bit fixed point with stochastic rounding is used for training while 16 bit fixed point is used for inference. In other words, a MAC can operate two pairs of 16bit operands or a pair of 32bit fixed point with stochastic rounding operands. It is synthesized in 15nm FinFet [101] to operate at 2.5GHz (Table 6.1).

Max Unit

Max unit receives one input stream and one big control signal to represent *new* stream of data. The length of stream is r^2, when r is pooling radius. At the end of stream, Max unit returns the maximum value of the stream and its ID (Fig. 2.7). It is only active for max-pooling layer feedforward path; therefore MAX comparator is designed to operate only 16bit precision.

Comparator Unit

Since MAX operation is required only for the max-pooling inference, 16 bit fixed point comparators are placed in PE. Based on pooling radius (r), r^2 data are streamed into controller, and the comparator unit returns the maximum value and its ID for backpropagation.

Table 6.2: Programming PMAG for Convolution and Fully connected layer

	7 level nested counters							Conv. In - Out				f (a,b,c,d)			
								p		q					
	R1	R2	R3	R4	R5	R6	R7	s	t	u	v	a	b	c	d
Conv-FF	N_O	H_O	W_O	N_I	D_K	H_K	W_K	r2	r6	r3	r7	r4	q	p	r5
Conv-BP	D_I	H_I	W_I	N_I	N_O	H_K	W_K	r2	r6	r3	r7	r4	q	p	r5
Conv-UP	1	N_I	H_O	W_O	D_I	H_K	W_K	r3	r6	r4	r7	q	p	r5	r2
FC-FF/BP (C. Vault)	H/P	W/L	P	L	K	1	1	-	-	-	-	r4	r2	r5	0
FC-FF/BP (I. vault)	H/P	W/L	P	L	K	1	1	-	-	-	-	r4	r3	r2	r1
FC-UP (C vault)	H/h	$\frac{W}{N_{MAC}}$	N_I	N_{MAC}	1	1	1	-	-	-	-	r4	r3	r2	r1
FC-UP (I. vault)	H/h	$\frac{W}{N_{MAC}}$	N_I	h	1	1	1	-	-	-	-	r4	r3	r2	r1

R1 $\sim$ R7: maximum value of r1 $\sim$ r7 loop. (minimum value are all zero)

Table 6.3: Programming PMAG for data rearranging and data preparation

	7 level nested counters			Conv. In - Out				f (a,b,c,d)				Two comparators			
				p		q									
	R1	R2	R3	s	t	u	v	a	b	c	d	h	H	k	K
Merge	D_I	PH_I	PW_I	-	-	-	-	r3	r2	r1	0	-	-	-	-
Partition	D_I	H_I	W_I	-	-	-	-	0	0	0	1	r2	$0 \sim PH_I$	r3	$0 \sim PW_I$
Add pad	D_I	PH_I	PW_I	r3	r	r2	r	p	q	r1	0	r3	$r \sim r + W_I$	r2	$r \sim r + H_I$
Remove pad	D_I	PH_I	PW_I	-	-	-	-	r3	r2	r1	0	r3	$r \sim r + W_I$	r2	$r \sim r + H_I$

R1 $\sim$ R3: maximum value of r1 $\sim$ r3 loop. (minimum value are all zero)
R4 $\sim$ R7: 1

Table 6.4: PE Program for computing operations

	Bit	CNT2	CNT1
Conv-FF	16	H_K, W_K	$W_K \times H_K$
Conv-BP	32	H_K, W_K	$W_K \times H_K$
Conv-UP	32	P, L	L
FC-FF	16	P, L	L
FC-BP	32	P, L	L
FC-UP	32	h	1

H_K, W_K: dimension of convolution kernels
P, L: dimension of partial matrix (Fig. 6.6)
h: length of partial vector (Fig. 6.7)

PE Operation

After two input buffers are filled (BUF Input 1 and BUF Input 2), BUF Input 1 pushes one 32bit input (one 32 bit operand or two 16 bit operands) while BUF Input 2 pushes k (N_{MAC}) 32bit inputs. For MAC operation (all cases except max pooling), k MAC arrays compute $\mathbf{y} = a\mathbf{x} + \mathbf{y}$, where **x** and **y** are vectors, which length is k (32bit) or $2k$ (16bit). For MAX operation (max pooling), k comparator returns max value as $\mathbf{y} = max(\mathbf{x}, \mathbf{y})$.

Convolution. In convolution, k inputs are processed by k MACs in parallel (SIMD level). Therefore, kernels are stored in BUF Input 1 and k inputs are stored in BUF Input 2. If k inputs cannot be stored in BUF Input 2 due to capacity issue, k subsets of k inputs are

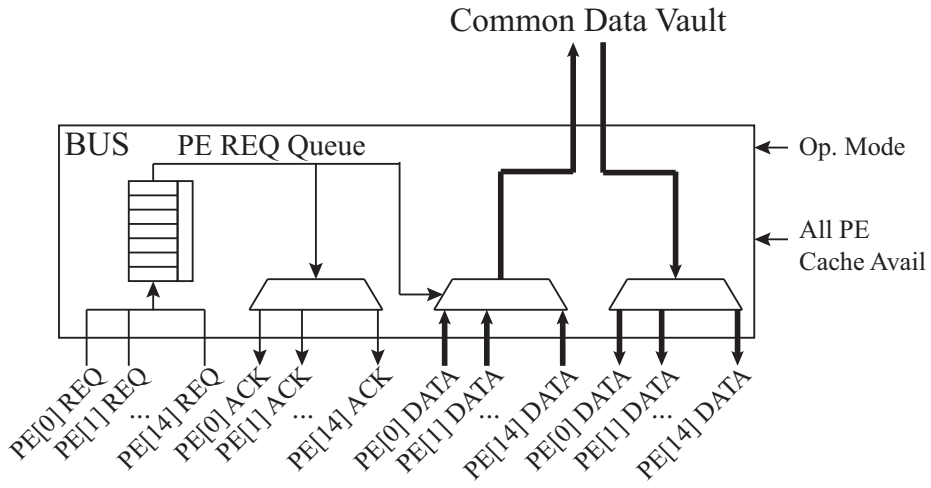


Figure 6.9: Block diagram of bus interface between state vault and all PEs.

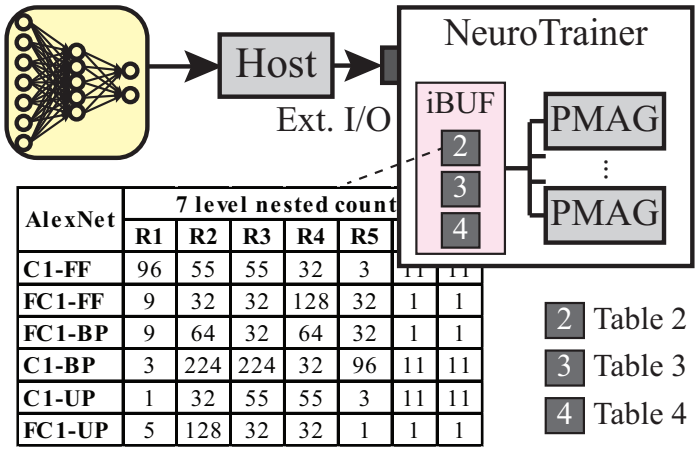
stored and newly required input ($k \times H_k$) is updated during the operation similar to [27]. For convolution backpropagation, W^T is easily obtained by sweeping counter values in CNT2 attached to BUF Input 1.

Matrix-Matrix multiplication. Similar to convolution, k inputs are processed in parallel (SIMD level). Therefore, partial weight matrix is loaded in BUF Input 1 and k partial inputs are stored in BUF Input 2. After consuming one partial weight matrix [a,b] (P by L in Fig. 6.6), next partial weight matrix [a,b+1] is processed. Similar to convolution, W^T is obtained by sweeping counter values in CNT2 attached to BUF Input 1.

Vector-Vector outer product. In fully connected update, k inputs cannot be processed in parallel. In computing AB^T, A is loaded in BUF Input 1 and B is loaded in BUF Input 2. In other words, k elements of B is delivered into k MAC units in a single clock (Fig. 6.7).

6.1.4 BUS Interface

Bus interface has two operation modes controlled by common data vault: broadcasting to all PEs and merging data into common data vault from all PEs. The BUS and PE communicates using three-way handshaking (REQ-ACK-SEND) for both operations. Broadcasting mode is set when all PE can take data (input buffer is ready) and during the broadcasting, any REQ from PE is ignored (broadcasting is prior to merging mode). During the merg-



AlexNet	7 level nested count						
	R1	R2	R3	R4	R5		
C1-FF	96	55	55	32	3	11	11
FC1-FF	9	32	32	128	32	1	1
FC1-BP	9	64	32	64	32	1	1
C1-BP	3	224	224	32	96	11	11
C1-UP	1	32	55	55	3	11	11
FC1-UP	5	128	32	32	1	1	1

Figure 6.10: Programming NeuroCube by host for given DNN.

ing mode, all PE send REQ and get ACK from the bus based on predetermined priorities among PEs. Although all 15 PEs request BUS for writing-back simultaneously, the impact of latency of entire writing back (for 15 PEs) on the throughput can be minimized as PE's computing latency dominates entire computing latency. The bus architecture is designed and synthesized to guarantee a bandwidth same as that of a single vault (10GBps). We use 4 stage pipe-lined BUS interface [109]; it takes 4 clock cycle between a vault to any PE.

6.2 Programming

Following the discussions in Section 6.1.2, Table 6.2 and Table 6.3 summarize the PMAG programming which includes setting range of 7 nested loops ($r1 \sim r7$) and connecting counter values to combination logic for different operations. For matrix-matrix multiplication (FC-FF/BP) and vector outer product (FC-UP), C.Vault is the programming value for PMAG attached to common data vault and I.Vault is the programming value for PMAG attached to independent data vault. Similar to PMAG, PE needs to be programmed to set: 1) operation type: MAC or MAX, 2) bit precision mode for MAC: 16 bit or 32 bit with/without SR , and 3) loop range for address generator for local buffers. Based on the discussions in Section 6.1.3, Table 6.4 summarizes the inputs for PE programming for different operations. In essence, the preceding three tables defines the instruction set architecture of

the NeuroCube.

Given a DNN, the host first generates the preceding three tables. Fig. 6.10 illustrates the programming process of the NeuroCube. To enable *autonomous operation* of the NeuroCube, we embed an on-chip instruction buffer (iBuffer) to store the preceding three tables (Figure 6.10(a)). Given a DNN, the host generates the preceding three tables and loads them in the iBuffer. During execution the layer-wise operation is controlled by the iBuffer (using a layer counter). To estimate the size of the iBuffer, consider that for a network with N layers, we need to program NeuroCube $\sim 4N$ times (Feedforward, backpropagation, weight update, data preparation if needed). Each time the amount of data for programming is 22Byte (18Byte for PMAG and 4Byte for PE). Therefore, a 16KB memory is sufficient as iBuffer and it can cover 186 layers. The latency of programming the iBuffer through HMC external interface is negligible compared to loading the input data.

6.3 Simulation Results

6.3.1 Performance Analysis

The performance of the NeuroCube is simulated using cycle-level simulator. All simulation results is based on minibatch size 32, which is recommended minimum size of minibatch [110]. All MACs, comparators, buffer in PE, BUS interface, PMAG are synthesized operate at 2.5GHz to maximize the single vault's bandwidth.

Fig. 6.11 shows throughput (TOPS/s) and latency (second) for a single input of each layer in Alexnet. For the one input image, inference took 0.31mS (3,228 images per second) and training took 1.97mS (507 images per second).

In feedforward phase, all convolution or fully connected layers shows similar throughput above 4.0 TOPS/s (4.2TOPS/s $\sim$ 4.7TOPS/s) which is close to the theoretical maximum for 16bit operation of our MAC (2.5GHz $\times$ 15 PEs $\times$ 32 MACs $\times$ 2 pairs inputs $\times$ 2 (multiplication and addition) = 4.8TOPS/s).

For backpropagation and weight update, 32bit with stochastic rounding is used. The-

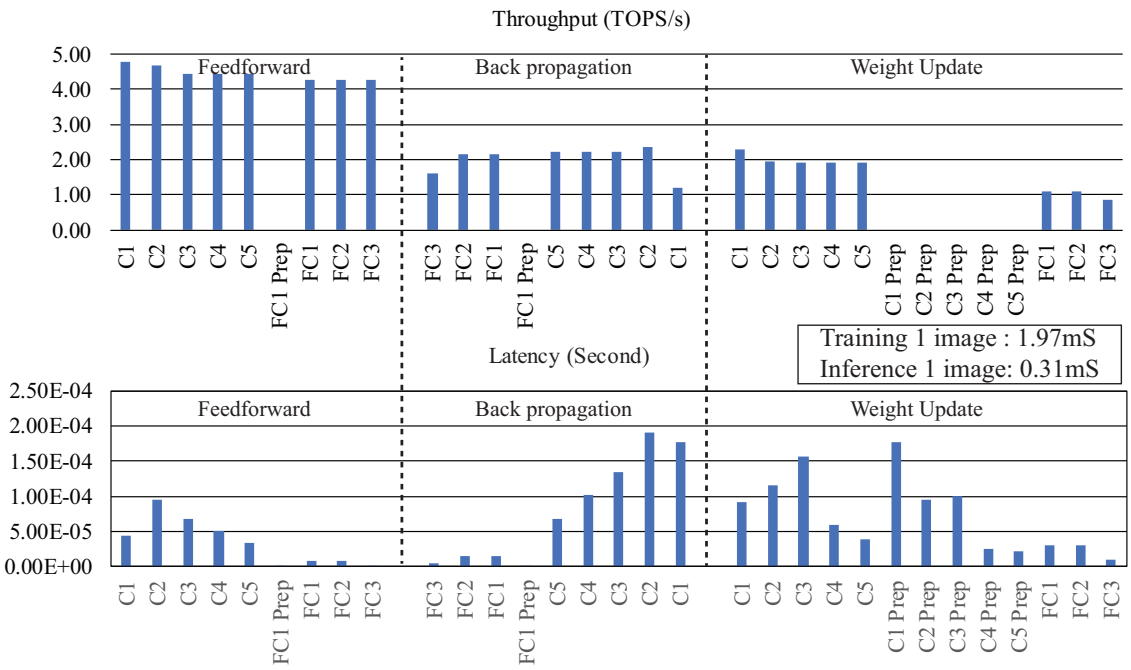


Figure 6.11: Simulation result for Alexnet in terms of latency (second) and throughput (Tera-Ops/sec: TOPS/s). C1 - C5: convolution layer, FC1 - FC3: fully connected layer, Prep: data preparation

oretical maximum throughput can be computed as 2.4 TOPS/s in the same manner. In backpropagation, FC3 (1.61 TOPS/s) and C1 layer (1.19 TOPS/s) show lower throughput than others. For FC3 backpropagation, the size of dY is not large enough to hide latency of *writing back* from all PEs to a single vault. In other words, the latency to generate output by iterating dY times is shorter than writing back the output to common data vault; writing back becomes bottleneck in the system. For C1 layer, input dimension is $55 \times 55 \times 96$ and kernel dimension is $11 \times 11 \times 3$. It can be processed as convolution since kernel size is small enough to fit in the local memory; but efficiently due to large kernel size compared to input.

In weight update, C1 $\sim$ C5 shows about 1.98 TOPS/s by translating convolution as matrix multiplication in large kernel case. However, FC layer (vector vector outer multiplication) shows about 1.02 TOPS/s, which is the worst case due to high network traffic between PE and independent vault since there is no data re-usage.

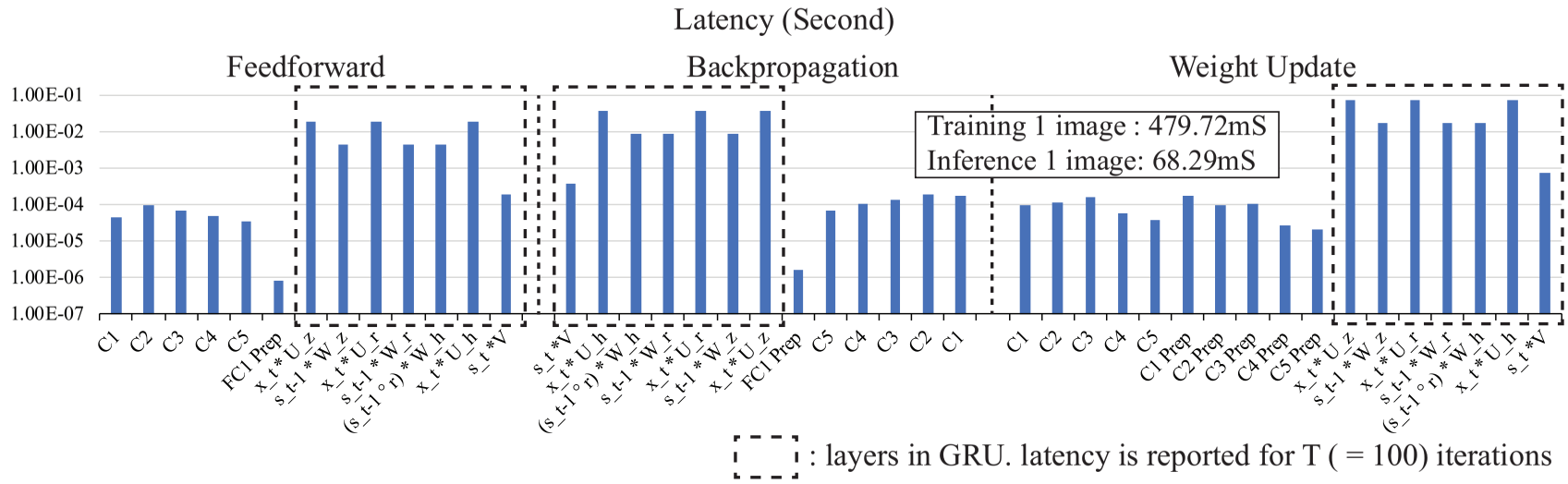


Figure 6.12: Latency analysis of each layer in DNN [42].

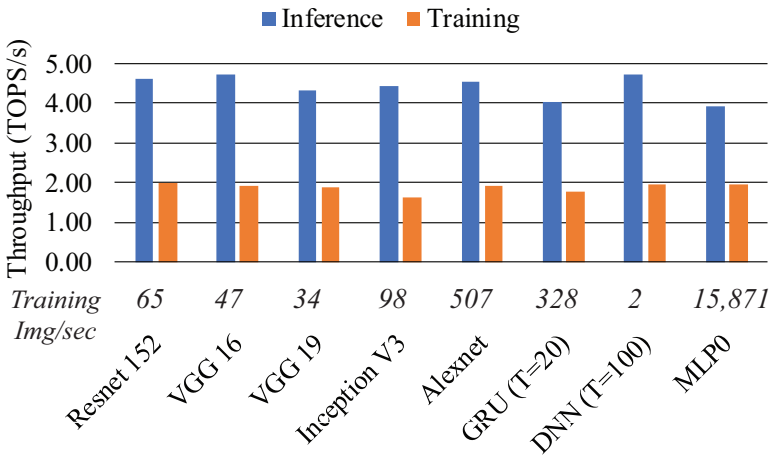


Figure 6.13: Simulation results for different benchmarks.

To see the performance of more complex and deeper network, we evalaute a DNN for generating image description [42], image feature extraction part is implemented as Alexnet and RNN (GRU) is attached after 5^{th} convolution layer (Fig. 5.11). A single GRU is composed of six fully connected layers for hidden neurons and one fully connected layer for output neurons. The number of input neurons in GRU is 43,264 and the number of hidden neurons in GRU is 10,000. We assume T for DNN is 100. Fig. 6.12 shows latency of each layer in DNN explained earlier. For the recurrent layers (in the dashed box), the latency is computed considering time windows (latency to across all time unfolded T layers); that's why it shows high latency than other layers.

Fig. 6.13 shows the throughput for various benchmarks including Resnet 152 [43], VGG 16, VGG 19 [45], Inception V3 [44], GRU [40], DNN for image description [42], and MLP0 [33] are also tested. Y-axis represent the throughput (TOPS/s) and the number on the X-axis represent the number of inputs can be trained in a second for each benchmark. For all benchmarks, inference shows $4.0 \sim 4.7$ TOPS/s and training shows 1.9 TOPS/s. Further, NeuroCube shows stable throughput (standard deviations less than 6% of average) for training with all benchmarks of varying complexity.

Table 6.5: Power and Area analysis of NeuroCube synthesized in 15nm FinFet [101].

	Area (mm^2)	**Power (W)**
PE	6.96E-02	1.55E-01
PMAG	2.00E-03	3.16E-03
Vault Ctrl.	7.73E-04	4.27E-03
32bit Bus	8.96E-03	3.70E-02
16KB: I-BUF	5.51E-03	1.02E-02

Logic die	1.17E+00	2.65E+00
4 DRAM dies		2.03E+00

6.3.2 Synthesis and Power Analysis

The computation fabric of the NeuroCube, including the PEs, bus interface, and PMAG with the vault controller is synthesized using 15nm FinFet [101].

As vault controller is a proprietary design, a 32bit SDRAM controller [109] is adopted as a reference vault controller. Table 6.5 summarizes power and area overhead of each module in the system. Total power consumption of logic layer is 2.64W and area overhead (including vault controller) is $1.17mm^2$. Even scaled up to compare with previous result synthesized in 28nm CMOS [60], total area is less than 5% of footprint of fabricated HMC ($68mm^2$). Average DRAM die power is computed during the simulation using 3.7pJ/bit from [60] and actual DRAM access pattern. The power densities of the logic die (0.039W/mm^2) and DRAM dies (0.030W/mm^2) in NeuroCube is well within the acceptable power densities ($1.5W/mm^2$, [68], of 3D stacked systems.

On average, NeuroCube consumes 4.64W and delivers 1.89 TFLOPS/s throughput and 406 GFLOPS/s/W of efficiency during training (32bit) while maintaining high training accuracy.

For HMC 2.0 [106], performance is estimated (Table **??**). With 32 vaults, 31 PEs can be placed; therefore throughput and logic power increases about twice. However, power of DRAM dies is same since total memory access is constant. Therefore, it shows 39% gain in efficiency.

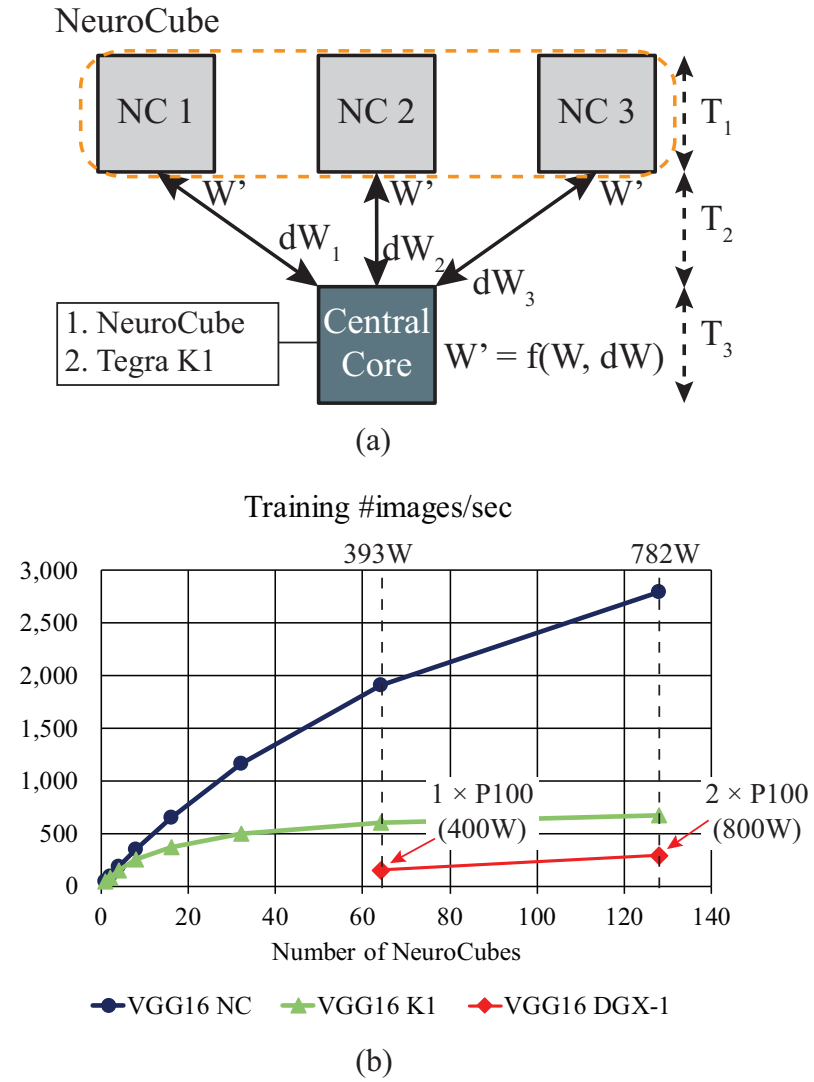


Figure 6.14: Scalability: (a) system of multiple NeuroCube. (b) Performance for VGG16 with central core being NeuroCube (VGG16 NC) and Tegra K1 (VGG16 K1).

6.3.3 Scalability to Multiple NeuroCubes

The multiple NeuroCube can be used in parallel for scalable training performance as illustrated in Fig. 6.14 (a). As all NeuroCube take same latency (T_1) for training a minibatch, we propose to perform **synchronous** training [49]. After training a single input batch, N NeuroCube delivers dW to a central unit (latency = T_2). The central unit needs to take all dW from N NeuroCube s, and generates new W (W') following: $W' = \eta \times average(dW) + W$, where η is learning rate. The above computation can be performed by another NeuroCube.

Table 6.6: Comparison with previous training accelerators

Work	NC*	NS [30]	SD [114]	GPU	CPU	This Work
Bit	16 FI	32 FL	16 FL / 32 FL	32 FL	32 FL	16 FI / 32 FI *
Node (nm)	15	28	14	16	32	15
Peak TFLOPS/s	0.13	0.96	1,400 / 680	51.24	1.60	4.4 (9.6) / 1.9 (4.1)
Power (W)	3.4	42.8	1,400	3200	145	4.7 (7.2)
Efficiency	38.8	22.5	331.7	16.01	11.0431	406 (566)

NC*: NeuroCube in Section 4, **NS**: NeuroStream, **SD**: ScaleDeep,
GPU: Nvida DGX-1 with 8 P100s, **CPU**: Intel E5-2699 v4
FL: floating point, FI: fixed point, FI*: fixed point with stochastic rounding
Efficiency: $GFLOPS/s/W$
For NT, () indicates *estimated* for HMC 2.0

However, to cover more generic approaches for weight update (e.g. AdaGrad [111] or Adam [112]), a software implementation, for example, using Tegra K1 (326 GFLOPS/s, 10W, 28nm) [113] can also be considered.

Fig. 6.14 (b) shows estimated training performance (number of images per second) of VGG 16 [45] by different number of NeuroCubes and two different types of central core.

Training performance using high-end GPU (NVIDIA DGX-1) is also reported. A single P100 in DGX-1 can train 150 images per second [46] with 400W power consumption. For the same power budget, 64 NeuroCube can operate in parallel and train 1,900 images delivering 13x speedup. The additional power consumption due to off-chip data movement estimated using HMC access energy of 10pJ/bit [60]. Ultimately, the performance scaling in NeuroCube is limited by the off-chip latency showing need for better system architecture and faster off-chip network.

6.4 Related Work

Table 6.6 compare NeuroCubes with previously reported DNN training accelerators. NeuroCube [28] and NeuroStream [30] presents inference engines using in-memory accelerators, which can also perform training. The results demonstrate higher efficiency over a GPU-baseline showing the promise of hardware acceleration. However, performance gain is nominal as no hardware was optimized for training.

Scaledeep [114] proposes specialized hardware for different computation kernels. A multi-chip module is synthesized using five different tiles (heterogeneous architecture) and allocating layers to different tiles based on their property (such as Byte/Ops). The design, composed of 7,032 tiles, demonstrates better power efficiency over GPUs.

The main difference between NeuroCube and Scale-Deep is the orthogonal approaches to optimize efficiencies of different kernel. Rather than changing a data flow in the hardware for different operations as performed in NeuroCube, ScaleDeep decides the tile distribution during design. Consequently, if the layer distribution in DNN architecture does not match the tile distributions, for example, if one kind of layer (convolution or fully connected) dominates the entire network, the tile utilization and efficiency is low. This effect is evident from [114] (see Fig. 20) which shows even for various CNN benchmarks, the standard deviation of efficiency is about 28% of average, which is expected to increase further if recurrent networks are considered.

In contrast, the NeuroCube uses a homogeneous architecture and dynamically changes the data flow and data mapping to optimize the performance of individual layers. The dynamic optimization, instead of design time decisions, allow NeuroCube to maintain similar throughput for much wider classes of benchmarks even including RNNs. The homogeneous architecture also makes the design easier to scale for parallel training. The secondary difference comes from the use of 3D in-memory acceleration in NeuroCube to reduce data movement power, and fixed point arithmetic with stochastic rounding for higher efficiency

(compared to floating point in ScaleDeep).

6.5 Conclusion

We have presented NeuroCube, an intelligent memory module with in-memory accelerators for energy-efficient training of different classes of DNNs. The NeuroCube utilizes a programmable data flow based execution model to optimize memory mapping and data re-use during different phases of training operation. The simulation results demonstrate potential for appreciable performance and power-efficiency gain over baseline GPU or alternative accelerators. The NeuroCube can form the building block of a scalable architecture for energy efficient training for deep neural networks. Ultimately, the performance scaling in a scalable training platform with NeuroCube is limited by the off-chip latency showing need for future research on better system architecture and faster off-chip network.

CHAPTER 7
CONCLUSION

In this thesis, digital deep learning accelerator, NeuroCube, for both inference and traininig and approximate computing for inference based on training conditions is presented.

To understand the impact of quantization error in deep learning algorith, the concept of approximate synapse for feedforward network is studied. The approximated synapses are selected based on gradient (error sensitivity of synaptic weights) precomputed during training phase. It is observed that training conditions play an important role in power saving during the inference with approximate computing. Further, it is shown that after the selection of approximate synapses, performing retraining can improve the accuracy; hence, more aggressive approximation be-comes possible for a target accuracy. Moreover, for training with high accuracy, fixed point with stochastic rounding is studied and low overhead design using a single pseudo random number generator is studied.

As a deep learning accelerator, NeuroCube is introduced as processor in memory architecture placing computing units below 3D stacked DRAM dies. Hybrid memory cube (HMC) from Micron is selected as 3D stacked memory platform due to its flexibility on the logic layer design. First version of NeuroCube is placing 4 by 4 2D array of processing elements, routers, and programmable address generator (PNG). Without a global controller, each PNG is programmed based on *predetermined* sequence of the operands (states or weights). Therefore, writing configurable registers in PNG allows NeuroCube to cover different types of layers: 2D convolution and fully connected layer. It is synthesized with 15nm FinFet process and shows there is no area or thermal violation of HMC packaging while achieve about 130GOPS/s for both convolution and fully connected layer.

To improve throughput of fully connected layer with many neurons (where input duplication is impractical), two data paths for weights and states are separated by allocating

a single vault for common data (state in fully connected layer). It can achieve more than 1,800 GOPS/s for dense connected layer ($\sim$13x compared to previous NeuroCube). It also provides efficient computing performance when some weights are zero (sparse global connection). The data flow based computing model provides the programmability for wide classes of DNNs while optimized data mapping provides significant performance gain. As the complexity and application of DNNs continue to grow, NeuroCube can provide the scalability, flexibility, and power efficiency necessary for future DNN accelerators.

As final verion of NeuroCube, it is designed to cover deep learning training as well. Training requires feedforward, backpropagation, and weight update. To maintain the accuracy, reconfigurable MAC units are used to operate 16bit fixed point in inference and 32bit fixed point with stochastic rounding in training. Rather than having different types of processing block in the single system (heterogeneous architecture), changing data movements and operation mode in the homogeneous architecture (cluster of same kind of PEs) for different layers or different operations. Compared to two previous ASIC accelerator for training, NeuroCube shows higher efficiency (GOPS/W) in both inference and training.

In conclusion, I proposed deep learning accelerator with data-flow based computing model embedded in 3D stacked DRAM. I believe deep learning acclerator with in the memory can provide higher throughput or better energy efficiency compared to GPU, which is the main computing fabric today, and it will be key factors to explode deep learning domain further. Since deep learning can allow small perturbation during the computing, approximate during the inference and training should be considered for accelerator in limited power budget system.

CHAPTER 8
FUTURE WORK

By placing computing units in the memory system, memory access overhead in terms of latency and energy can be reduced significantly. However, processor in memory system has a limitiation in area and thermal constraints and it can be problem in placing huge PEs in a NeuroCube. Detail study about NeuroCube with different 3D stacked DRAMs (HBM [56]) should be studied.

Next bottlenecks of deep learning system are i) data movement between multiple processing elements in the NeuroCube and ii) interface between multiple NeuroCubes on the board level. Data movement in the NeuroCube can be minimized by data reusage or placing big buffer in the system. However, board level interconnect overhead is critical to design the system with multiple NeuroCubes or system with more dense storage unit (SSD, HDD) for traning data. Fast and reliable communication between NeuroCubes remains as future work.

Since there is no *golden* methods of numeric precision in deep learning, more flexible computing unit is required. Future deep learning accelerator should be able to cover very low precision in inference (binary neural network [115]) or very high precision (double floating point) for training. In NeuroCube, 16 or 32 bit fixed points are tried (two scenarios). While maintaining high data bandwidth, maintaining high utilization for those reconfigurable PEs for different precision size remains as future work.

Since stacking DRAM dies on the logic die is heterogenous packaging, advanced memory cell can be fabricated separately and stacked on top of the logic die. Therefore, NeuroCube design with ReRAM [116] or STT-RAM [117] remains as future work.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems, 2012*, pp. 1097–1105.

[2] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, IEEE, 2014, pp. 1725–1732.

[3] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition, 2015*, pp. 1–9.

[4] P. Stone and M. Veloso, "Multiagent systems: A survey from a machine learning perspective," *Autonomous Robots*, vol. 8, no. 3, pp. 345–383, 2000.

[5] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[6] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International Conference on Machine Learning*, 2016, pp. 173–182.

[7] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, ACM, 2008, pp. 160–167.

[8] *Obama Political Speech generator - Recurrent Neural Network*, `https://github.com/samim23/obama-rnn/`.

[9] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[10] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[11] S. Hochreiter, Y. Bengio, P. Frasconi, *et al.*, *Gradient flow in recurrent nets: The difficulty of learning long-term dependencies*, 2001.

[12] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, and F. Kawsar, "An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices," in *Proceedings of the 2015 International Workshop on Internet of Things towards Applications*, ACM, 2015, pp. 7–12.

[13] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *Information Processing in Sensor Networks (IPSN), 2016 15th ACM/IEEE International Conference on*, IEEE, 2016, pp. 1–12.

[14] T. J. Chainer, M. D. Schultz, P. R. Parida, and M. A. Gaynes, "Improving data center energy efficiency with advanced thermal management," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 2017.

[15] Y. LeCun, L. Jackel, L. Bottou, *et al.*, "Learning algorithms for classification: A comparison on handwritten digit recognition," *Neural networks: the statistical mechanics perspective*, vol. 261, p. 276, 1995.

[16] Y. Taigman, M. Yang, M. Ranzato, *et al.*, "Deepface: Closing the gap to human-level performance in face verification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1701–1708.

[17] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, *et al.*, "Progressive neural networks," *arXiv preprint arXiv:1606.04671*, 2016.

[18] Y. Bengio, J. Louradour, R. Collobert, *et al.*, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning*, ACM, 2009, pp. 41–48.

[19] `https://www.tensorflow.org/performance/benchmarks/`.

[20] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An fpga-based processor for convolutional networks," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, IEEE, 2009, pp. 32–37.

[21] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, IEEE, 2011, pp. 109–116.

[22] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on*, IEEE, 2014, pp. 696–701.

[23] F. Conti and L. Benini, "A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, EDA Consortium, 2015, pp. 683–688.

[24] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, IEEE, 2014, pp. 609–622.

[25] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, ACM, 2015, pp. 199–204.

[26] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," *arXiv preprint arXiv:1602.01528*, 2016.

[27] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, IEEE, 2016, pp. 367–379.

[28] D. Kim, J. Kung, S. Chai, *et al.*, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, IEEE, 2016, pp. 380–392.

[29] M. Gao, J. Pu, X. Yang, *et al.*, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2017, pp. 751–764.

[30] E. Azarkhish, D. Rossi, I. Loi, *et al.*, "Neurostream: Scalable and energy efficient deep learning with smart memory cubes," *arXiv preprint arXiv:1701.06420*, 2017.

[31] E. Nurvitadhi, J. Sim, D. Sheffield, *et al.*, "Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic," in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, EPFL, 2016, pp. 1–4.

[32] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "14.2 dnpu: An 8.1 tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks," in *Solid-State Circuits Conference (ISSCC), 2017 IEEE International*, IEEE, 2017, pp. 240–241.

[33] N. P. Jouppi, C. Young, N. Patil, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th International Symposium on Computer Architecture*, IEEE Press, 2017.

[34] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, pp. 1–42, 2014.

[35] S. S. Haykin, S. S. Haykin, S. S. Haykin, and S. S. Haykin, *Neural networks and learning machines*. Pearson Education Upper Saddle River, 2009, vol. 3.

[36] F. Rosenblatt, "Principles of neurodynamics. perceptrons and the theory of brain mechanisms," DTIC Document, Tech. Rep., 1961.

[37] H. Jaeger, *Tutorial on training recurrent neural networks, covering bppt, rtrl, ekf and the" echo state network" approach*. German National Research Center for Information Technology, 2002.

[38] S. Gould, R. Fulton, and D. Koller, "Decomposing a scene into geometric and semantically consistent regions," in *Computer Vision, 2009 IEEE 12th International Conference on*, IEEE, 2009, pp. 1–8.

[39] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[40] J. Chung, C. Gulcehre, K. Cho, *et al.*, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[41] P. J. Werbos, "Backpropagation through time: What it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[42] A. Karpathy and L. Fei-Fei, "Deep visual-semantic alignments for generating image descriptions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 3128–3137.

[43] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[44] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826.

[45] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[46] `http://dlbench.comp.hkbu.edu.hk/`.

[47] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[48] R. A. Dunne and N. A. Campbell, "On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function," in *Proc. 8th Aust. Conf. on the Neural Networks, Melbourne, 181*, vol. 185, 1997.

[49] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "Firecaffe: Near-linear acceleration of deep neural network training on compute clusters," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2592–2600.

[50] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[51] J. Sim, J.-S. Park, M. Kim, D. Bae, Y. Choi, and L.-S. Kim, "14.6 a 1.42 tops/w deep convolutional neural network recognition processor for intelligent ioe systems," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, IEEE, 2016, pp. 264–265.

[52] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, IEEE, 2016, pp. 262–263.

[53] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR, abs/1510.00149*, vol. 2, 2015.

[54] *DDR3 SDRAM, JESD79-3F*, `http://www.jedec.org/standards-documents/docs/jesd-79-3d`.

[55] *WIDE I/O 2, JESD229-2*.

[56] *High Bandwith Memory, JESD235*, `http://www.jedec.org/standards-documents/results/jesd235`.

[57] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile dram," in *ACM SIGARCH Computer Architecture News*, IEEE Computer Society, vol. 40, 2012, pp. 37–48.

[58] Hybrid Memory Cube Consortium, *Hybrid memory cube specification 1.0*, 2013.

[59] P. Rosenfeld, "Performance exploration of the hybrid memory cube," PhD thesis, University of Maryland, 2014.

[60] J. Jeddeloh and B. Keeth, "Hybrid memory cube new dram architecture increases density and performance," in *2012 Symposium on VLSI Technology (VLSIT)*, 2012.

[61] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong, "25.2 a 1.2 v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, IEEE, 2014, pp. 432–433.

[62] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "High performance axi-4.0 based interconnect for extensible smart memory cubes," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, EDA Consortium, 2015, pp. 1317–1322.

[63] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ACM, 2015, pp. 105–117.

[64] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ACM, 2015, pp. 336–348.

[65] J. Zhao, G. Sun, G. H. Loh, and Y. Xie, "Optimizing gpu energy efficiency with 3d die-stacking graphics memory and reconfigurable memory interface," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, p. 24, 2013.

[66] E. Azarkhish, D. Rossi, I. Loi, *et al.*, "Design and evaluation of a processing-in-memory architecture for the smart memory cube," in *International Conference on Architecture of Computing Systems*, Springer, 2016, pp. 19–31.

[67] M. Amir, D Kim, J Kung, D Lie, S Yalamanchili, and S Mukhopadhyay, "Neurosensor: A 3d image sensor with integrated neural accelerator," in *SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S), 2016 IEEE*, IEEE, 2016, pp. 1–2.

[68] W. Huang, M. R. Stan, S. Gurumurthi, *et al.*, "Interaction of scaling trends in processor architecture and cooling," in *Semiconductor Thermal Measurement and Management Symposium, 2010. SEMI-THERM 2010. 26th Annual IEEE*, IEEE, 2010, pp. 198–204.

[69] J. Yang, X. Zeng, and S. Zhong, "Computation of multilayer perceptron sensitivity to input perturbation," *Neurocomputing*, vol. 99, pp. 390–398, 2013.

[70] J. Y. Choi and C.-H. Choi, "Sensitivity analysis of multilayer perceptron with differentiable activation functions," *IEEE Transactions on Neural Networks*, vol. 3, no. 1, pp. 101–107, 1992.

[71] A. F. Murray and P. J. Edwards, "Enhanced MLP performance and fault tolerance resulting from synaptic weight noise during training," *IEEE Transactions on Neural Networks*, vol. 5, no. 5, pp. 792–802, 1994.

[72] J. H. Kung, "Energy-efficient digital hardware platform for learning complex systems," PhD thesis, Georgia Institute of Technology, 2017.

[73] A. W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study," *IEEE Transactions on Neural Networks*, vol. 18, no. 1, pp. 240–252, 2007.

[74] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "AxNN: Energy-efficient neuromorphic systems using approximate computing," in *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), 2014*, pp. 27–32.

[75] J. Kung, D. Kim, and S. Mukhopadhyay, "A power-aware digital feedforward neural network platform with backpropagation driven approximate synapses," in *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), 2015*, pp. 85–90.

[76] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices - ASPLOS '14*, vol. 49, pp. 269–284.

[77] U. Lotrič and P. Bulić, "Applicability of approximate multipliers in hardware neural networks," *Neurocomputing*, vol. 96, pp. 57–65, 2012.

[78] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *EDAA Design, Automation & Test in Europe, 2014*, p. 95.

[79] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy, "Multiplier-less Artificial Neurons exploiting error resiliency for energy-efficient neural computing," in *EDAA Design, Automation & Test in Europe, 2016*, pp. 145–150.

[80] G. Srinivasan, P. Wijesinghe, S. S. Sarwar, A. Jaiswal, and K. Roy, "Significance driven hybrid 8T-6T SRAM for energy-efficient synaptic storage in artificial neural networks," in *EDAA Design, Automation & Test in Europe, 2016*, pp. 151–156.

[81] C. Schuldt, I. Laptev, and B. Caputo, "Recognizing human actions: A local SVM approach," in *IEEE Internation Conference on Pattern Recognition, 2004*, vol. 3, pp. 32–36.

[82] A. Nagendran, D. Harper, and M. Shah, "New system performs persistent wide-area aerial surveillance," *SPIE Newsroom*, vol. 5, pp. 20–28, 2010.

[83] M. Zhang and A. A. Sawchuk, "USC-HAD: A daily activity dataset for ubiquitous activity recognition using wearable sensors," in *ACM Conference on Ubiquitous Computing, 2012*, pp. 1036–1043.

[84] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[85] J. Kung, D. Kim, and S. Mukhopadhyay, "Dynamic Approximation with Feedback Control for Energy-Efficient Recurrent Neural Network Hardware," in *IEEE/ACM International Symposium on Low Power Electronics and Design, 2016*, pp. 168–173.

[86] X. Liu, S. Li, K. Fang, Y. Ni, Z. Li, and Y. Deng, "Radixboost: A hardware acceleration structure for scalable radix sort on graphic processors," in *IEEE International Symposium on Circuits and Systems (ISCAS), 2015*, pp. 1174–1177.

[87] *MNIST dataset*.

[88] *CNAE-9 dataset*.

[89] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," *CoRR*, vol. abs/1606.06160, 2016.

[90] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," *arXiv preprint arXiv:1312.5851*, 2013.

[91] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[92] M. Stevenson, R. Winter, and B. Widrow, "Sensitivity of feedforward neural networks to weight errors," *IEEE Transactions on neural networks*, vol. 1, no. 1, pp. 71–80, 1990.

[93] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, pp. 1737–1746.

[94] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, IEEE Press, 2016, pp. 267–278.

[95] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance fpga-based accelerator for large-scale convolutional neural networks," in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, EPFL, 2016, pp. 1–9.

[96] T. Na, J. H. Ko, J. Kung, and S. Mukhopadhyay, "On-chip training of recurrent neural networks with limited numerical precision," in *Neural Networks (IJCNN), 2017 International Joint Conference on*, IEEE, 2017, pp. 3716–3723.

[97] L. Cavigelli, M. Magno, and L. Benini, "Accelerating real-time embedded scene labeling with convolutional networks," in *Proceedings of the 52nd Annual Design Automation Conference*, ACM, 2015, p. 108.

[98] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," *Neural computation*, vol. 12, no. 10, pp. 2451–2471, 2000.

[99] L. O. Chua and L. Yang, "Cellular neural networks: Theory," *Circuits and Systems, IEEE Transactions on*, vol. 35, no. 10, pp. 1257–1272, 1988.

[100] *Synopsys 32/28nm Generic Library*, `https://www.synopsys.com/COMMUNITY/UNIVERSITYPROGRAM/Pages/32-28nm-generic-library.aspx`.

[101] *Nangate FreePDK15 Open Cell Library*, `http://www.nangate.com/?page_id=2328`.

[102] E. Karl, Z. Guo, J. W. Conary, J. L. Miller, Y.-G. Ng, S. Nalam, D. Kim, J. Keane, U. Bhattacharya, and K. Zhang, "0.6 v 1.5 ghz 84mb sram design in 14nm finfet cmos technology," in *Solid-State Circuits Conference-(ISSCC), 2015 IEEE International*, IEEE, 2015, pp. 1–3.

[103] http://www.itrs.net/.

[104] A. Sridhar, A. Vincenzi, D. Atienza, and T. Brunschwiler, "3d-ice: A compact thermal model for early-stage design of liquid-cooled ics," *Computers, IEEE Transactions on*, vol. 63, no. 10, pp. 2576–2589, 2014.

[105] W. J. Song, S. Mukhopadhyay, and S. Yalamanchili, "Energy introspector: A parallel, composable framework for integrated power-reliability-thermal modeling for multicore architectures," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, IEEE, 2014, pp. 143–144.

[106] Hybrid Memory Cube Consortium, *Hybrid memory cube specification 2.0*, 2014.

[107] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," *Diploma, Technische Universität München*, p. 91, 1991.

[108] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "Cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[109] *OpenCores*, http://http://opencores.org/.

[110] http://svail.github.io/rnn_perf/.

[111] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[112] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[113] T. NVIDIA, "K1: A new era in mobile computing," *Nvidia, Corp., White Paper*, 2014.

[114] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ACM, 2017, pp. 13–26.

[115] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[116] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[117] K. Wang, J. Alzate, and P. K. Amiri, "Low-power non-volatile spintronic memory: Stt-ram and beyond," *Journal of Physics D: Applied Physics*, vol. 46, no. 7, p. 074 003, 2013.